Algorithmes compensés en arithmétique flottante : précision, validation, performances

Nicolas Louvet

Université de Perpignan Via Domitia
Laboratoire ELIAUS :
Électronique, Informatique, Automatique et Systèmes

THÈSE

soutenue le 27 novembre 2007 par

Nicolas Louvet

pour l'obtention du grade de

Docteur de l'université de Perpignan
spécialité : Informatique

au titre de l'école doctorale ED 305

Algorithmes compensés en arithmétique flottante : précision, validation, performances.

Commission d'examen :

Nicholas J. HIGHAM	Professeur, University of Manchester	Rapporteur
Jean-Michel MULLER	Directeur de recherches, CNRS, ENS de Lyon	Rapporteur
Jean-Marie CHESNEAUX	Professeur, Université Pierre et Marie Curie	Examinateur
Bernard GOOSSENS	Professeur, Université de Perpignan	Examinateur
Siegfried M. RUMP	Prof. Dr., Hamburg University of Technology	Examinateur
Philippe LANGLOIS	Professeur, Université de Perpignan	Directeur

Table des matières

Remerciements

En premier lieu, je tiens à remercie mon directeur de thèse Philippe Langlois pour son encadrement attentif, ses conseils avisés et son enthousiasme tout au long de ces trois années. Je le remercie également pour ses nombreux encouragements et ses relectures patientes de mon manuscrit, qui ont contribuées à en améliorer la rédaction sous bien des aspects.

Je remercie Nick Higham et Jean-Michel Muller d'avoir accepté de rapporter sur cette thèse, malgré les délais très brefs que je leur imposais, et de m'avoir permis, au travers de leurs commentaires, d'améliorer la qualité de ce document. Je remercie de plus Nick Higham pour m'avoir invité à passer quelques jours à Manchester pour me faire part de ces remarques.

Je remercie également Jean-Marie Chesneaux pour m'avoir fait l'honneur de présider mon jury, ainsi que Bernard Goossens et Siegfried M. Rump pour m'avoir fait celui d'y prendre part. J'ai particulièrement apprécié l'intérêt qu'ils ont porté à l'égard de mon travail de thèse, notamment au travers des nombreuses remarques qu'ils ont pu faire au cours de ma soutenance.

Un grand merci à tous les membres de l'équipe DALI, qui m'ont beaucoup aidé et encouragé tout au long de ma thèse. Merci donc à Marc Daumas, David Defour, Bernard Goossens, Philippe Langlois, Matthieu Martel, Christophe Nègre et David Parello. Merci à mes co-bureaux Vincent Beaudenon, Sylvain Collange, Pascal Giorgi et Stef Graillat qui ont su me supporter au quotidien ! Merci également à Mourad Bouache, Ali El Moussaoui et Benaoumer Senouci pour leurs encouragements dans la dernière ligne droite.

Je tiens à remercier particulièrement Bernard pour les nombreux conseils qu'il m'a apporté au cours de ces trois années, d'une part en ce qui concerne mon entrée dans le monde de l'enseignement, mais aussi à propos de mon travail de thèse.

Je remercie David Defour pour ses conseils éclairés, ainsi que Pascal et David Parello qui ont su prendre le temps de me faire part de leur propre expérience au cours de discussions interminables ! Merci aussi à Christophe pour m'avoir souvent épaulé dans mes calculs alambiqués !

Un grand merci à Stef, dont l'expérience en tant que thésard à l'Université de Perpignan

m'a beaucoup profité ! Je remercie aussi Stef pour les nombreuses discussions scientifiques que nous avons eu, et pour sa collaboration dans nos travaux communs.

Je remercie aussi David Parello et Bernard pour m'avoir aidé pour la rédaction de certaines parties de cette thèse touchant à leur spécialité, l'architecture des ordinateurs.

Je remercie également les membres du LP2A, devenu laboratoire ELIAUS, qui m'ont offert un cadre de travail agréable et convivial pendant trois ans.

Je tiens également à remercier les membres de l'équipe Arénaire, qui m'ont toujours très bien accueilli lors de mes différentes visites, et qui, je n'en doute pas, feront de même au cours de mon Postdoc ! Un grand merci en particulier à Nathalie et Gilles qui ont rendu ce Postdoc possible.

Je remercie chaleureusement Gilles Dequen, mon directeur de stage de DEA à Amiens, en particulier pour m'avoir très bien conseillé dans la poursuite de mon parcours.

Enfin, je remercie mes parents, ma famille, ainsi que mes plus proches amis pour leur soutien et leur compréhension. Je tiens également à exprimer une pensée sympathique pour tous les amis avec lesquels j'ai pu lier connaissance à Perpignan et dont le soutien m'a beaucoup aidé, en particulier Mathieu, Stef, Fred, Julie, Aurélie, Virginie, David, Anne-Lyse, Marie, Adama... et bien d'autres !

Chapitre 1

Introduction

Plan du chapitre : Dans ce chapitre introductif, nous présentons en Section 1.1 les motivations justifiant ce travail, puis nos principales contributions en Section 1.2. Une présentation détaillée de cette thèse est effectuée chapitre par chapitre à la Section 1.3.

1.1 Motivations

De nombreux schémas numériques, utilisés pour le calcul scientifique ou en ingénierie, sont mis en œuvre en précision finie, généralement grâce à l'arithmétique flottante disponible sur les ordinateurs modernes. De tels schémas ne sont souvent qu'approchés et peuvent dans certains cas fournir des résultats très imprécis, voir aberrants. En particulier, un résultat calculé en précision finie peut ne comporter aucun chiffre exact.

Lorsque l'on s'intéresse à la résolution d'un problème, de nature physique ou économique par exemple, à l'aide de méthodes numériques, on distingue généralement quatre types d'erreurs.

1. Les erreurs *de modèle* prennent en compte le fait que le problème auquel on s'intéresse est formulé à l'aide d'un modèle mathématique qui ne rend pas nécessairement compte exactement de la réalité.
2. Les erreurs *de troncature* sont introduites par la méthode numérique que l'on applique au modèle mathématique. C'est la cas par exemple lorsqu'intervient dans le modèle mathématique une fonction analytique représentée par un développement en série : cette série sera nécessairement tronquée, de manière à ce que l'on puisse en calculer une valeur approchée.
3. Les erreurs *de données* sont dues au fait que les données du problème auquel on s'intéresse proviennent de mesures physiques imprécises, ou encore au fait qu'il a fallu les arrondir avant de les stocker en mémoire.
4. Les erreurs *d'arrondi* sont inhérentes au calcul en précision finie. L'ensemble des réels est en effet approché à l'aide d'un nombre fini de valeurs, dans le cas qui nous intéresse particulièrement, un ensemble de nombres flottants. Chaque opération arithmétique effectuée est donc entachée d'une erreur d'arrondi.

Dans cette thèse, nous nous concentrons sur l'effet des erreurs d'arrondi en arithmétique flottante, sans prendre en compte les autres sources d'erreurs décrites ci-dessus. À elles seules, les erreurs d'arrondi sont en effet susceptibles de dégrader considérablement, voire

totalement, la précision d'une solution calculée en arithmétique flottante. Ce phénomène est bien connu, et sera illustré au travers de plusieurs exemples au cours des chapitres suivants. Évoquons néanmoins deux raisons pour lesquelles l'effet des erreurs d'arrondi peut se révéler particulièrement néfaste.

D'une part, l'effet des erreurs d'arrondi peut être amplifié de part la nature même du problème numérique considéré. Ici intervient la notion de nombre de conditionnement, qui permet de quantifier la difficulté de résoudre un problème précisément, indépendamment de l'algorithme numérique utilisé.

De plus, la puissance des ordinateurs actuels, en termes de calcul flottant, permet d'envisager la résolution numérique de problèmes de dimension de plus en plus importante. Même si chaque opération arithmétique n'introduit qu'une faible erreur d'arrondi, comment s'assurer de la fiabilité d'une simulation enchaînant plusieurs milliards d'opérations flottantes ?

Notre problématique sera donc la suivante : **comment améliorer et valider la précision d'un résultat calculé en arithmétique flottante, tout en conservant de bonnes performances pratiques ?**

Dans cette thèse, nous nous intéresserons à cette problématique en nous plaçant dans le cadre de l'arithmétique flottante telle que définie par la norme IEEE-754, et nous étudierons en particulier deux exemples.

1. Nous considérerons le problème de l'**évaluation polynomiale**, qui apparaît dans de nombreux domaines du calcul scientifique.
2. Nous étudierons également celui de la résolution de **systèmes linéaires triangulaires**, qui constitue un algorithme de base de l'algèbre linéaire numérique.

Dans les deux cas, nous avons utilisé la **compensation des erreurs d'arrondi** comme principale technique afin d'améliorer la précision de la solution calculée.

Dans la suite de cette section, nous précisons les trois axes complémentaires de notre problématique : amélioration de la précision du résultat, validation et performances.

1.1.1 Améliorer la précision du résultat

Une réponse possible pour améliorer la précision d'un résultat calculé par un algorithme en arithmétique flottante est d'augmenter la précision des calculs. Mais que faire si la meilleure précision disponible $\mathbf{u}$, sur l'architecture sur laquelle on travaille, ne permet pas d'assurer une précision satisfaisante du résultat ?

Supposons que les caractéristiques de l'arithmétique flottante disponible soient régies par la norme IEEE-754. La meilleure précision standardisée de manière portable par cette norme est la double précision. Dans le mode d'arrondi au plus proche, la double précision permet des calculs en précision $\mathbf{u} = 2^{-53} \approx 10^{-16}$, soit une mantisse d'environ 16 chiffres décimaux.

Parmi les solutions logicielles permettant d'augmenter la précision de calcul au delà de la double précision IEEE-754, citons l'exemple de l'arithmétique double-double. Un double-double est la somme non évaluée de deux flottants double précision IEEE-754.

L'arithmétique double-double permet d'effectuer les opérations élémentaires sur ce type de données, et simule ainsi deux fois la double précision IEEE-754, soit la précision $\mathbf{u}^2$.

Dans [57], la nécessité de calculs intermédiaires en précision double de la précision courante est illustrée sur de nombreux exemples où sont utilisés les algorithmes de base de l'algèbre linéaire numérique : résolution de systèmes linéaires, raffinement itératif et problème des moindres carrés sont quelques-un de ces exemples. L'arithmétique double-double est ainsi utilisée pour le développement des *Extended and Mixed precision BLAS* [56, 57].

L'approche consistant à augmenter la précision des calculs n'est cependant pas la seule : on peut également ré-écrire les algorithmes, de manière à *compenser* les erreurs d'arrondi.

En effet, à l'aide d'algorithmes que nous désignerons par la suite sous le nom de *transformations exactes* (de l'anglais *error-free transformation* [70]), il est possible de calculer l'erreur d'arrondi générée par chaque opération flottante.

Étant donné un algorithme, le principe de la *compensation des erreurs d'arrondi* est d'adjoindre à cet algorithme le calcul des erreurs d'arrondi qu'il génère, et de les utiliser afin de corriger le résultat calculé.

De nombreux exemples d'algorithmes compensés sont connus dans la littérature, et montrent qu'il est ainsi possible d'améliorer la précision du résultat calculé. Citons les exemples suivants :

- sommation simplement compensée : Kahan (1965) [45], Møller (1965) [65, 64], puis Pichat (1972) [78, 79], Neumaier (1974) [67], et Ogita, Rump et Oishi (2005) [70] ;
- sommation doublement compensée : Priest (1992) [82] ;
- produit scalaire compensé (2005) [70].

Citons également la méthode CENA, développée par Langlois [49], qui permet d'automatiser la compensation des erreurs d'arrondi.

Dans cette thèse, nous poursuivons l'étude de l'amélioration de la précision par compensation des erreurs d'arrondi, au travers des deux exemples précédemment cités : évaluation de polynômes et résolution de systèmes triangulaires.

1.1.2 Valider la qualité du résultat calculé

Même si l'on est parvenu à améliorer la précision d'un algorithme, le résultat calculé est toujours susceptible d'être entaché d'une certaine erreur. Il est donc préférable qu'un algorithme calcule, en même temps que le résultat attendu, une borne d'erreur permettant d'en garantir la qualité.

Nous parlerons *d'algorithme validé* (de l'anglais *self-validating* [83, 84]) ou encore *d'algorithme produisant un résultat validé* pour désigner un algorithme qui fournit une garantie sur la qualité du résultat qu'il calcule. Citons ici la définition suivante, proposée par Rump [83] :

> The basic principle of "verification algorithms" or "algorithms with validated results", also called "algorithms with automatic verification" is as follows. First, a pure floating point algorithm is used to compute an approximate solution for a given problem. This approximation is, hopefully, of good quality ; however, no quality assumption is used at all. Second, a final verification step is appended. After this step, either error bounds are computed for the previously calculated

> approximation or, an error message signals that error bounds could not be computed.

L'une des méthodes les plus utilisées pour valider le résultat d'un calcul numérique est l'arithmétique d'intervalle [84, 44]. L'utilisation des modes d'arrondi dirigés prévus par la norme IEEE-754 permet également, indépendamment de l'arithmétique d'intervalle, de procéder à la validation de certains algorithmes [73, 85].

Ici, nous avons privilégié l'approche consistant à n'utiliser que l'arithmétique flottante disponible, dans le mode d'arrondi au plus proche, afin de valider nos algorithmes compensés, comme dans les références [70, 76]. Cette approche permet notamment d'obtenir d'excellentes performances en pratique, et d'assurer une bonne portabilité des algorithmes.

1.1.3 Performance des algorithmes compensés

Maintenir de bonnes performances pratiques tout en augmentant la précision est aujourd'hui l'un des grands challenges en calcul scientifique.

L'efficacité pratique des algorithmes compensés a déjà été observée dans [70]. Comme nous l'avons déjà rappelé ci-dessus, les auteurs de cet article étudient les algorithmes de sommation et de produit scalaires compensés. Ils comparent notamment les performances pratiques du produit scalaire compensé Dot2 à son homologue en arithmétique double-double issu de la bibliothèque XBLAS [56, 57], et désigné sous le nom de DotXBLAS. Comme cela est prouvé dans [70], précisons que les algorithmes Dot2 et DotXBLAS fournissent des résultats de précision équivalente.

Les tests de performance reportés dans [70] montrent que le produit scalaire compensé Dot2 s'exécute environ deux fois plus rapidement que l'algorithme DotXBLAS. Des résultats similaires sont également reportés dans le cadre d'une comparaison entre la sommation compensée et de son homologue issu de la bibliothèque XBLAS.

D'une façon générale, les bonnes performances des algorithmes compensés, face à leur équivalents basés sur des solutions génériques pour augmenter la précision de calcul, justifient également l'intérêt que nous leur portons dans cette thèse.

D'autre part, les tests de performances effectués dans [70] montrent que les décomptes d'opérations flottantes ne sont pas forcément suffisants pour comprendre les temps d'exécution mesurés sur les architectures modernes. Dans cet article, l'efficacité de Dot2 est effectivement constatée, mais elle reste inexpliquée. Nous nous intéresserons donc également à la question suivante : comment expliquer les excellentes performances pratiques des algorithmes compensés sur les architectures modernes ?

1.2 Nos contributions

Nous résumons ci-dessous nos contributions selon les trois axes de notre problématique.

Amélioration de la précision du résultat.

Nous nous intéressons en premier lieu à l'évaluation de polynômes univariés à coefficients flottants. Nous proposons une version compensée du schéma de Horner : la précision du résultat compensé produit par cet algorithme est la même que s'il avait été calculé par le

schéma de Horner classique avec une précision interne doublée. Nous donnons en outre une condition suffisante, portant sur le nombre de conditionnement de l'évaluation, permettant d'assurer que le résultat produit par le schéma de Horner compensé est un arrondi fidèle du résultat exact. Par arrondi fidèle, nous entendons ici que le résultat compensé est soit égal à $p(x)$, soit l'un des deux flottants encadrant ce résultat exact.

En généralisant ensuite cet algorithme, nous proposons une version du schéma de Horner dans laquelle les erreurs d'arrondis sont compensées $K-1$ fois : nous décrivons ainsi un schéma d'évaluation produisant un résultat aussi précis que s'il avait été calculé par le schéma de Horner classique en K fois la précision de travail ($K \geq 2$).

Nous étudions également la résolution de systèmes linéaires triangulaires : il s'agit de l'un des problèmes de base en algèbre linéaire numérique, résolu classiquement par l'algorithme de substitution. Nous montrons comment compenser les erreurs d'arrondi générées par l'algorithme, et mettons en évidence les liens qui existent entre cette méthode et le fait d'utiliser une étape de raffinement itératif [39, chap. 12] en précision doublée pour améliorer la précision de la solution compensée. Nos expériences numériques indiquent que la solution compensée est à nouveau aussi précise que si elle avait été calculée en précision doublée, puis arrondi vers la précision de travail. Notre analyse d'erreur ne permet pas de garantir ce comportement numérique, mais indique que la précision de la solution compensée est effectivement améliorée.

Validation de la qualité du résultat.

Au travers de l'exemple du schéma de Horner compensé, nous montrons comment valider la précision du résultat compensé. Nous proposons le calcul d'une borne d'erreur *a posteriori* pour le résultat de l'évaluation compensée, ne reposant que sur des opérations élémentaires en arithmétique flottante, dans le mode d'arrondi au plus proche : afin d'obtenir un code portable et efficace, nous n'utilisons ni bibliothèque de calcul d'intervalles, ni changement du mode d'arrondi. Nous proposons également un test permettant de déterminer dynamiquement si l'évaluation compensée produit un arrondi fidèle de l'évaluation exacte.

La borne d'erreur *a posteriori* est effectivement bien plus fine que celle obtenue via l'analyse d'erreur *a priori* du schéma de Horner. Précisons que cette borne d'erreur *a posteriori* est elle-même sujette aux erreurs d'arrondi. Nous prendrons donc soigneusement en compte les erreurs d'arrondi commises lors de son évaluation, de manière à prouver rigoureusement qu'il s'agit bien d'une borne sur l'erreur directe, et non pas d'une simple estimation.

En utilisant les mêmes techniques, nous procédons également à la validation du schéma de Horner compensé $K-1$ fois.

Dans les deux cas, nos tests de performances montrent que le coût de la validation de l'évaluation compensée est en pratique très faible : le temps d'exécution moyen observé n'est qu'au plus 1.5 fois plus élevé que celui de l'algorithme non validé.

Performances des algorithmes compensés.

Nos mesures de performances justifient pleinement l'intérêt pratique des algorithmes compensés face aux autres solutions logicielles permettant de simuler une précision équivalente. Notamment, dans le cas d'un doublement de la précision, les algorithmes compensés

décrits dans cette thèse s'exécutent toujours au moins deux fois plus rapidement que leurs homologues basés sur l'arithmétique double-double, pourtant considérée comme une référence en terme de performance [57].

Les processeurs superscalaires modernes sont conçus pour tirer partie du parallélisme d'instructions présent dans les programmes. Grâce à une étude détaillée, nous montrons clairement que le schéma de Horner compensé présente plus de parallélisme d'instructions que le schéma de Horner basé sur l'arithmétique double-double.

Chaque opération arithmétique faisant intervenir un double-double se termine par une étape de renormalisation [81, 57]. Nous montrons également que le défaut de parallélisme d'instructions dans le schéma de Horner avec les double-double est précisément dû à ces étapes de renormalisation.

Les conclusions de cette étude se généralisent facilement aux autres algorithmes compensés étudiés dans cette thèse : l'absence d'étapes de renormalisation dans ces algorithmes est la raison pour laquelle ils présentent plus de parallélisme d'instructions que leurs équivalents en arithmétique double-double, ce qui explique de façon qualitative leurs bonnes performances pratiques.

1.3 Présentation détaillée de la thèse

Le chapitre 2 constitue une brève introduction à l'analyse d'erreur en arithmétique flottante. On y introduit brièvement les notions de base permettant de comprendre les effets de la précision finie sur les calculs numériques : erreur inverse, erreur directe et nombre de conditionnement. Nous rappelons les principales caractéristiques de l'arithmétique flottante, ainsi que les notations classiques. Ces notations sont illustrées sur l'exemple de l'évaluation polynomiale par le schéma de Horner qui sera largement étudié dans les chapitres suivants. Nous passons également en revue les principales bibliothèques logicielles permettant de simuler une précision de travail arbitraire, et introduisons plus en détail la notion d'algorithme compensé.

Au chapitre 3, nous passons en revue les transformations exactes connues pour les opérations élémentaires $+, -, \times$ et $/$ en arithmétique flottante, en présentant aussi le cas de l'opération « Fused-Multiply-and-Add » (FMA). Ces transformations exactes constituent les briques de base pour la conception d'algorithmes compensés. Nous présenterons ensuite deux applications connues de ces transformations exactes. Nous rappellerons en effet les résultats obtenus par Ogita, Rump et Oishi à propos de la sommation et du produit scalaire compensés [70], puis nous donnerons quelques détails sur les arithmétiques double-double et quad-double.

1.3.1 Schéma de Horner compensé (chapitre 4)

Motivation : Développer des algorithmes numériques rapides et fiables pour l'évaluation polynomiale reste un grand challenge. Les méthodes de calcul numérique de racines se basent en général sur des méthodes itératives de type Newton, qui nécessitent d'évaluer un polynôme et ses dérivées. Higham [39, chap. 5] consacre, par exemple, un chapitre entier aux polynômes et en particulier à l'évaluation polynomiale.

Le schéma de Horner est un algorithme connu pour sa stabilité numérique [39, p.95][20]. Néanmoins, même lorsque l'on considère un polynôme p à coefficients flottants, le résultat de l'évaluation de $p(x)$ par le schéma de Horner peut être arbitrairement moins précis que l'unité d'arrondi $\mathbf{u}$. C'est le cas lorsque le nombre de conditionnement $\mathrm{cond}(p, x)$ de l'évaluation polynomiale est grand devant $\mathbf{u}^{-1}$: on parle alors d'évaluation mal conditionnée.

Contribution : Nous proposons dans ce chapitre une alternative à l'utilisation de l'arithmétique double-double dans le contexte de l'évaluation polynomiale. Le principe du schéma de Horner compensé que nous présentons ici est celui de la compensation des erreurs d'arrondi générées par le schéma de Horner classique, à l'aide des transformations exactes décrites au chapitre 3.

Nous démontrons, à l'aide d'une analyse d'erreur *a priori*, que le résultat de l'évaluation compensé est aussi précis que celui calculé par le schéma de Horner classique en précision doublée, avec un arrondi final vers la précision de travail. En outre, nous montrons que tant que le conditionnement est petit devant $\mathbf{u}^{-1}$, le schéma de Horner compensé calcule un arrondi fidèle du résultat exact.

Les résultats évoqués jusqu'à présent sont valables tant qu'aucun dépassement de capacité n'intervient au cours des calculs. Nous fournirons également dans ce chapitre une borne d'erreur *a priori* pour le résultat calculé par le schéma de Horner compensé, satisfaite aussi en présence d'underflow.

Ce travail sur le schéma de Horner compensé a fait l'objet d'une publication dans les actes de la conférence IEEE ARITH18 [52].

1.3.2 Performances du schéma de Horner compensé (chapitre 5)

Motivation : Le doublement de la précision des calculs obtenu à l'aide du schéma de Horner compensé (algorithme CompHorner) introduit nécessairement un surcoût en comparaison du schéma de Horner classique (algorithme Horner). Définissons ce surcoût comme étant égal au ratio du temps d'exécution de CompHorner sur le temps d'exécution de Horner. Il est clair que ce surcoût dépend de l'architecture utilisée pour effectuer la mesure.

Mais il est commun de penser que le surcoût introduit par CompHorner doit être sensiblement égal au ratio du nombre d'opérations flottantes exécutées par CompHorner sur le nombre d'opérations flottantes effectuées par Horner : on devrait alors s'attendre à un surcoût de l'ordre de 11. Or, au travers de mesures effectuées sur un certain nombre d'architectures récentes, nous verrons que ce surcoût est en pratique de l'ordre de 3, ce qui est donc bien inférieur à celui que l'on aurait pu attendre au regard du décompte des opérations flottantes. Nous observons aussi ce phénomène dans le cas du schéma de Horner basé sur les double-doubles (algorithme DDHorner). Néanmoins, nos expériences montrent que CompHorner s'exécute en pratique environ 2 fois plus rapidement que DDHorner.

Comment expliquer le faible surcoût introduit par le schéma de Horner compensé par rapport au schéma de Horner classique, ainsi que ses bonnes performances face au schéma de Horner basé sur l'arithmétique double-double ?

Contribution : Dans ce chapitre, nous proposons une analyse détaillée du parallélisme d'instructions disponible dans les algorithmes CompHorner et DDHorner. Nous quantifions le nombre moyen d'instructions de chacun de ces algorithmes qui peuvent théoriquement être exécutées simultanément sur un processeur idéal. Ce processeur idéal, défini dans [33, p.240], est un processeur dans lequel toutes les contraintes artificielles sur le parallélisme d'instructions sont supprimées. Dans ce contexte, l'IPC (« instruction per clock ») théorique est environ 6 fois meilleur pour l'algorithme de Horner compensé que pour son équivalent en arithmétique double-double.

Chaque opération arithmétique faisant intervenir un double-double se termine par une étape de renormalisation [81, 57]. Nous montrons également que le défaut de parallélisme d'instructions dans DDHorner est précisément dû à ces étapes de renormalisation. Autrement dit, l'absence de telles étapes de renormalisation dans CompHorner explique le fait qu'il présente plus de parallélisme d'instructions que DDHorner, et par conséquent son efficacité pratique sur les architectures superscalaires modernes.

Ces travaux ont fait l'objet d'un rapport de recherche en 2007 [53], soumis au journal IEEE *Transactions on Computers*.

1.3.3 Amélioration du schéma de Horner compensé avec un FMA (chapitre 6)

Motivation : L'instruction « Fused-Multiply-and-Add » (FMA) est disponible sur certains processeurs modernes, comme par exemple le Power PC d'IBM ou l'Itanium d'Intel. Rappelons brièvement son principe : étant donnés trois flottants a, b et c, l'instruction FMA calcule $a \times b + c$ avec seulement une erreur d'arrondi [61].

Dans le contexte de l'évaluation polynomiale, le FMA permet d'exécuter le schéma de Horner plus rapidement, et en diminuant de moitié le nombre d'erreurs d'arrondi générées. D'une manière générale, lorsque l'on travaille sur une architecture disposant d'un FMA, on a tout intérêt à adapter nos algorithmes de manière à tirer profit de cette instruction. Dans notre contexte, qui est celui des algorithmes compensés, on peut envisager d'utiliser l'instruction FMA de deux manières distinctes.

Premièrement, comme nous le verrons au chapitre 3, le FMA permet d'effectuer la transformation exacte d'une multiplication très efficacement, en seulement deux opérations flottantes [46, 61, 68]. On peut donc tirer partie de l'instruction FMA pour compenser plus efficacement les erreurs d'arrondi commises lors des multiplications flottantes.

D'autre part, une transformation exacte pour le FMA a été proposée par Boldo et Muller [12]. En voici le principe : étant donnés trois flottants a, b et c, $\mathsf{ThreeFMA}(a, b, c)$ produit trois flottants x, y, et z tels que $a \times b + c = x + y + z$, avec $x = \mathsf{FMA}(a, b, c)$. La transformation exacte ThreeFMA peut donc être utilisée pour compenser les erreurs d'arrondi entachant les opérations FMA.

Contribution : Dans ce chapitre, nous envisageons ces deux alternatives, et décrivons deux versions du schéma de Horner compensé adaptées pour tirer partie de l'instruction FMA. Nous décrivons :

- l'algorithme $\mathsf{CompHorner}_{\mathsf{fma}}$, dans lequel les erreurs d'arrondi sur les multiplications effectuées par le schéma de Horner classique sont compensées à l'aide du FMA ;

– l'algorithme CompHornerFMA, qui est une version compensée du schéma de Horner avec FMA, les erreurs d'arrondi générées étant calculées grâce à ThreeFMA.

Pour ces deux algorithmes, nous avons obtenu respectivement deux nouvelles bornes d'erreur, plus fines que celle de l'algorithme initial CompHorner. Ces améliorations de la borne d'erreur dans le pire cas s'avèrent toutefois trop faibles pour être observées dans nos expériences numériques.

Néanmoins, nous montrons que ces deux améliorations du schéma de Horner compensé s'exécutent toutes deux significativement plus rapidement que le schéma de Horner basé sur l'arithmétique double-double, qui pourtant tire également partie de la présence d'un FMA. De plus, nous verrons que $\mathsf{CompHorner}_{\mathsf{fma}}$ est en pratique l'alternative la plus efficace pour simuler une précision de travail doublée en présence d'un FMA.

L'étude présentée dans ce chapitre montre donc qu'il est préférable de tirer parti de la présence du FMA afin de compenser les erreurs d'arrondi générées par les multiplications, plutôt que de compenser les erreurs d'arrondi générées par le FMA à l'aide de ThreeFMA.

Ce travail a été publiée en partie dans les actes de la conférence ACM SAC 2006 [28]. Une version étendue a été publiée dans les actes faisant suite à la conférence SCAN 2006 [54].

1.3.4 Validation de l'évaluation polynomiale compensée (chapitre 7)

Motivation : Comme nous le verrons au travers d'expériences numériques, l'erreur directe entachant le résultat compensé calculé par CompHorner est largement surestimée par la borne d'erreur *a priori* démontrée au chapitre 4. D'une manière générale, le caractère pessimiste des bornes d'erreurs *a priori* obtenues à partir du modèle standard de l'arithmétique flottante est un phénomène bien connu, et relaté notamment par Higham [39, p.65]. De plus, la borne *a priori* que nous démontrons pour le schéma de Horner compensé fait intervenir le résultat exact $p(x)$ de l'évaluation polynomiale, qui est par définition inconnu : cette borne ne peut donc pas être évaluée en pratique.

Contribution : Nous proposons dans ce chapitre une version validée de l'évaluation polynomiale compensée : nous montrons comment calculer une borne d'erreur *a posteriori* pour le résultat compensé, effectivement bien plus fine que la borne *a priori* du chapitre 4. L'algorithme validé CompHornerBound ne repose que sur des opérations élémentaires en arithmétique flottante, dans le mode d'arrondi au plus proche : nous n'utilisons ni bibliothèque de calcul d'intervalles, ni changement du mode d'arrondi.

L'évaluation de cette borne d'erreur *a posteriori* est elle-même sujette aux erreurs d'arrondi, comme la plupart des calculs effectués en arithmétique flottante. Nous prendrons donc aussi en compte les erreurs d'arrondi commises lors de son évaluation, de manière prouver rigoureusement qu'il s'agit bien d'une borne sur l'erreur directe, et non pas d'une simple estimation.

Les tests de performances que nous effectuons montrent que le coût de la validation de l'évaluation compensée est en pratique très faible : le temps d'exécution moyen observé pour CompHornerBound n'est qu'au plus 1.3 fois celui de CompHorner.

Ces travaux concernant la validation de l'évaluation compensée ont été publiés dans les actes de la conférence IEEE ARITH18 [52].

1.3.5 Schéma de Horner compensé $K-1$ fois (chapitre 8)

Motivation : Rappelons que le résultat calculé par l'algorithme CompHorner présenté au chapitre 4 est aussi précis que s'il avait été calculé par le schéma de Horner en précision doublée $\mathbf{u}^2$, avec un arrondi final vers la précision de travail $\mathbf{u}$. En particulier, la précision du résultat compensé est de l'ordre de l'unité d'arrondi tant que le conditionnement est petit devant $\mathbf{u}^{-1}$. Par contre, lorsque $\mathrm{cond}(p,x)$ est plus grand que $\mathbf{u}^{-2}$, le résultat calculé par CompHorner ne présente plus aucun chiffre exact.

Contribution : Au chapitre 8, nous décrivons un nouvel algorithme compensé, que nous nommons CompHornerK, permettant d'effectuer une évaluation polynomiale en K fois la précision de travail ($K \geq 2$). Via une analyse d'erreur *a priori* nous montrons que le résultat compensé calculé par CompHornerK est aussi précis que s'il avait été calculé par le schéma de Horner classique en précision $\mathbf{u}^K$, puis arrondi vers la précision $\mathbf{u}$. Nous procéderons également à la validation de l'algorithme CompHornerK, en appliquant le même type de technique qu'au chapitre 7.

Il est à noter qu'une précision de travail $\mathbf{u}^K$ peut également être simulée à l'aide de bibliothèques génériques. C'est le cas par exemple de la bibliothèque flottante multiprécision MPFR [31], qui permet de travailler en précision arbitraire, et donc en particulier en précision $\mathbf{u}^K$. Citons également la bibliothèque quad-double [35], qui permet de quadrupler la double précision de IEEE-754, et donc de simuler une précision de travail de l'ordre de 212 bits. Nous utiliserons également cette bibliothèque pour évaluer comparativement les performances de l'algorithme CompHornerK, dans le cas particulier où $K = 4$.

Les tests de performance que nous avons effectués permettent à nouveau de justifier l'intérêt pratique de l'algorithme CompHornerK : ceux-ci montrent en effet que cet algorithme s'exécute significativement plus rapidement que les alternatives basées sur les bibliothèques génériques indiquées ci-dessus, tant que $K \leq 4$. L'algorithme que nous proposons ici pourra donc être utilisé avec avantage pour doubler, tripler ou quadrupler la précision de travail.

1.3.6 Résolution compensée de systèmes triangulaires (chapitre 9)

Motivation : La résolution de systèmes linéaires triangulaires est l'un des problèmes de base en algèbre linéaire numérique, résolu classiquement à l'aide de l'algorithme de substitution [39, chap. 8]. Tout comme le schéma de Horner dans le cas de l'évaluation polynomiale, l'algorithme de substitution est connu pour sa stabilité numérique [39, chap. 8]. Néanmoins, précision de la solution calculée par substitution d'un système triangulaire $Tx = b$ peut être arbitrairement mauvaise, puisqu'elle est proportionnelle au conditionnement de Skeel $\mathrm{cond}(T,x)$ du système, défini par

$$\mathrm{cond}(T,x) = \frac{\| |T^{-1}||T||x| \|_\infty}{\|x\|_\infty}.$$

qui est naturellement non borné. L'algorithme de substitution est notamment implanté en arithmétique double-double dans la bibliothèque XBLAS [56], afin de permettre la résolution de systèmes triangulaires mal conditionnés [57].

Contribution : Nous étudions dans ce chapitre l'effet de la compensation des erreurs d'arrondi dans le cas de la résolution d'un système triangulaire $Tx = b$ par l'algorithme de substitution. On suppose que T est une matrice triangulaire à coefficient flottants, et que le second membre b est également à coefficients flottants. Notons $\widehat{x}$ la solution du système $Tx = b$ calculée en arithmétique flottante par substitution.

Nous montrons que le principe du calcul d'une solution compensée $\overline{x}$ du système est en fait équivalent à l'utilisation d'une étape de raffinement itératif [89, 63, 8, 38, 25] pour améliorer la précision de la solution initiale $\widehat{x}$. Notons $r = b - T\widehat{x}$, le résidu exact associé à la solution $\widehat{x}$. La seule latitude dont on dispose quant au calcul d'une solution compensée $\overline{x}$ réside dans le choix d'une méthode pour le calcul d'une valeur approchée $\widehat{r}$ du résidu r.

Nous proposons donc une méthode de calcul du résidu basée sur l'utilisant des transformations exactes pour les opérations élémentaires : il est en effet possible d'exprimer r en fonction des erreurs d'arrondi générées par l'algorithme de substitution, ce qui permet de calculer un résidu approché $\overline{r}$. Le résidu approché ainsi obtenu est aussi précis que s'il avait été calculé en précision doublée, ce qui est une pratique courante lorsque l'on utilise le raffinement itératif [39, chap. 12].

En utilisant cette méthode de calcul d'un résidu approché à l'aide des transformations exactes, nous déduisons un algorithme de substitution compensé que nous appelons CompTRSV. Nos expériences numériques indiquent que la solution compensée $\overline{x}$ produite par CompTRSV est aussi précise que si elle avait été calculée en précision doublée, puis arrondie vers la précision de travail.

L'analyse d'erreur que nous effectuons de l'algorithme CompTRSV ne permet pas de prouver ce comportement numérique. En effet, la meilleure borne que nous déterminons sur la précision de la solution compensé ne fait pas intervenir le conditionnement de Skeel du système $Tx = b$, mais le facteur

$$\mathrm{K}(T, x) := \frac{\| (|T^{-1}||T|)^2 |x| \|_\infty}{\|x\|_\infty},$$

qui est un majorant de $\mathrm{cond}(T, x)$. Au travers de nos expériences numériques nous verrons comment interpréter cette borne d'erreur.

En outres, nos tests de performances montrent que CompTRSV s'exécute sur les architectures modernes au moins deux fois plus rapidement que la fonction de résolution de systèmes triangulaires de la bibliothèque XBLAS, tout en simulant également en pratique un doublement de la précision des calculs. Cela justifie à nouveau l'intérêt pratique de l'algorithme compensé CompTRSV face à son équivalent en arithmétique double-double.

Une partie de ces travaux a été publiée dans les actes de la conférence IMACS 2005 [51].

1.3.7 Conclusions, perspectives et annexes

Après avoir rappelé les principaux résultats obtenus, nous présentons quelques perspectives de travaux futurs. Dans cette thèse, nous proposons de nombreux résultats de tests de performances qui mettent en évidence l'efficacité pratique des algorithmes compensés sur les architectures modernes. Nous présentons la synthèse de ces résultats expérimentaux sous la forme de graphiques ou de tableaux de moyennes au long des chapitres 5, 6, 7, 8 et 9. Nous incluons donc en annexe les résultats de nos mesures de performances, ainsi qu'une présentation détaillée des environnements expérimentaux utilisés.

Chapitre 2

Analyse d'erreur et arithmétique flottante

Plan du chapitre : En section Section 2.2, nous rappellerons brièvement quelques notions sur l'analyse d'erreur en précision finie. Nous passerons en revue les principales caractéristiques du calcul en arithmétique flottante à la Section 2.3, et présenterons les notations utiles. Nous montrerons comment ces notations s'appliquent dans le cas du schéma de Horner en Section 2.4. Nous passons également en revue les principales bibliothèques logicielles permettant de simuler une précision de travail arbitraire, et introduisons plus en détail la notion d'algorithme compensé en Section 2.5.

2.1 Introduction

Nous commençons par une brève introduction à l'analyse d'erreur en précision finie, plus particulièrement dans le cadre de l'arithmétique flottante. Nous nous sommes largement inspiré des références [39, chap. 1],[50, chap. 1] et [20, chap. 1].

Nous montrons en quoi consiste l'analyse d'erreur en précision finie, notamment au travers des notions d'erreur directe et inverse, de stabilité des algorithmes et de nombre de conditionnement.

Nous passerons ensuite en revue les principales caractéristiques du calcul à l'aide des nombres flottants, et introduirons le modèle standard de l'arithmétique flottante. Nous nous intéresserons en particulier au cas de l'arithmétique IEEE-754 [41], largement disponible sur les ordinateurs modernes. Nous introduirons également les notations utilisées spécifiquement pour l'analyse d'erreur en arithmétique flottante.

Ces notations seront utilisées pour l'analyse d'erreur du schéma de Horner. Cette application classique nous permettra d'illustrer les notions vues précédemment, et sera également souvent utilisée dans le cadre de l'étude du schéma de Horner compensé, aux chapitres 4 à 7.

Nous citerons ensuite les principales bibliothèques logicielles permettant de simuler une précision de travail arbitraire. Nous introduirons également la notion d'algorithme compensé, qui sera largement illustrée au cours des chapitres suivants.

2.2 Analyse d'erreur en précision finie

Nous rappelons d'abord la distinction classique entre précision des calculs et précision de la solution. Nous définissons également les notions d'erreur inverse, d'erreur directe et de nombre de conditionnement. Nous avons choisi de les présenter très simplement en considérant le cas d'une fonction réelle de la variable réelle. Insistons néanmoins sur le fait que ces définitions se généralisent bien entendu lorsque l'on s'intéresse à des fonctions d'un espace vectoriel quelconque vers un autre : il suffit alors de choisir les normes adaptées — voir par exemple [15].

2.2.1 Notions de précision

Soit $\hat{x}$ une approximation d'un nombre réel x non nul. Les deux principales façons de mesurer la précision de $\hat{x}$ sont l'*erreur absolue*

$$E_a(\widehat{x}) = |\widehat{x} - x|,$$

et l'*erreur relative*

$$E_r(\widehat{x}) = \frac{|\widehat{x} - x|}{|x|},$$

qui ne peut être définie que si $x \neq 0$. Dans le cas où x et $\widehat{x}$ sont deux vecteurs de même dimension, les notions d'erreur absolue et d'erreur relative s'étendent à l'aide d'une norme adéquate $\|\cdot\|$, et deviennent respectivement

$$E_a(\widehat{x}) = \|\widehat{x} - x\|, \quad \text{et} \quad E_r(\widehat{x}) = \frac{\|\widehat{x} - x\|}{\|x\|}.$$

Il est important de distinguer la *précision d'un résultat* calculé de la *précision de calcul*, La précision d'un résultat fait référence à l'erreur relative entachant une quantité calculée, approchant un certain résultat exact, et résultant généralement d'un algorithme de calcul.

La *précision de calcul*, que nous appellerons également *précision de travail* ou encore *précision courante*, désigne l'erreur relative commise lors de chaque opération arithmétique élémentaire $+, -, \times$ ou $/$.

En arithmétique flottante, la précision de chaque opération arithmétique est majorée par l'*unité d'arrondi* notée $\mathbf{u}$. En effet, considérons deux nombres flottants a et b, ainsi qu'une opération arithmétique $\circ = +, -, \times, /$. En notant $x = a \circ b$ le résultat exact de l'opération, et $\widehat{x} = \mathrm{fl}(a \circ b)$ le résultat calculé en arithmétique flottante, alors on a

$$E_r(\widehat{x}) = \frac{|\widehat{x} - x|}{|x|} \leq \mathbf{u}.$$

L'unité d'arrondi sera donc généralement confondue avec la précision de calcul.

2.2.2 Erreur directe et erreur inverse

Considérons f une fonction réelle de la variable réelle. On suppose de plus que f est définie par la donnée d'un procédé de calcul dans lequel tous les calculs sont réputés exacts. On peut par exemple considérer la fonction f définie par $f : x \mapsto x^2 + x - 1$.

En pratique, le problème du calcul de $f(x)$ sera souvent résolu à l'aide d'un algorithme numérique, effectuant en précision finie, et dans un certain ordre, les opérations définissant f. En raison des erreurs commises lors des calculs intermédiaires en précision finie, et des erreurs de données, cet algorithme numérique ne réalise pas la fonction f, mais une fonction $\widehat{f}$: l'algorithme calcule ainsi une approximation $\widehat{y} = \widehat{f}(x)$ du réel $y = f(x)$.

Le but de l'*analyse d'erreur directe* est de majorer, ou plus rarement d'estimer, la distance séparant le résultat calculé $\widehat{x}$ du résultat exact x, qui sera par conséquent appelé *erreur directe*. Ce type de majoration peut être obtenu en propageant les erreurs générées par chaque opération effectuée par l'algorithme numérique étudié. L'erreur directe peut être majorée de façon relative ou absolue, selon que l'on considère $E_a(\widehat{y})$ ou $E_r(\widehat{y})$.

On retiendra donc que l'analyse d'erreur directe apporte des éléments de réponse à la question « Quelle est la précision du résultat calculé par l'algorithme numérique considéré ? ». De plus, il est important de noter que l'analyse d'erreur directe ne permet pas de différencier l'influence du problème de celle de l'algorithme, quant à la précision du résultat calculé.

Plutôt que de chercher à majorer l'erreur directe entachant $\widehat{y} = \widehat{f}(x)$, on peut dans un premier temps tenter de répondre à la question « Pour quel jeu de données le problème a t'il effectivement été résolu ? ». Il s'agit précisément du but de l'*analyse d'erreur inverse*, dans laquelle on cherche à identifier le résultat calculé $\widehat{y}$ à l'évaluation exacte de f en une donnée perturbée $x + \Delta x$, de manière à ce que l'on ait

$$\widehat{y} = f(x + \Delta x), \tag{2.1}$$

tout en bornant la perturbation Δx [89, 39].

Une fois l'analyse d'erreur inverse menée, on pourra généralement en déduire une majoration de l'erreur directe. Citons ici Wilkinson [89, chap. 1, §6] :

> In practice we have found that the backward analysis is often much the simpler, particulary in connexion with floating-point computation. It may well be objected that backward analysis is incomplete since ultimately we must be concerned with the difference between the *computed* solution to a problem and its *exact* solution. This is indeed true and there is always a final stage at which this difference must be assessed.

En général, plus d'une perturbation Δx satisfait l'équation (2.1). On définit donc l'*erreur inverse* associée à la solution approchée $\widehat{y}$ du problème considéré comme égale à la valeur minimale de $|\Delta x|$, sur l'ensemble des perturbation Δx satisfaisant la relation (2.1). La relation entre erreur inverse et erreur directe est illustrée par le diagramme classique de la figure 2.1.

Un algorithme pour le calcul d'une valeur approchée de $y = f(x)$ est dit *inverse stable* si, pour toute donnée x, le résultat $\widehat{y} = \widehat{f}(x)$ calculé par cet algorithme présente une erreur inverse « petite ». Un algorithme inverse stable fournit donc la solution exacte du problème pour un jeu de données entachées de « petites » perturbations.

La définition de ce qu'est une erreur inverse « petite » dépend bien entendu du contexte dans lequel on se place. En général, on considère un algorithme comme inverse stable dès lors que l'erreur inverse qu'il provoque est du même ordre de grandeur que l'incertitude présente sur les données. En arithmétique flottante, un algorithme est considéré comme

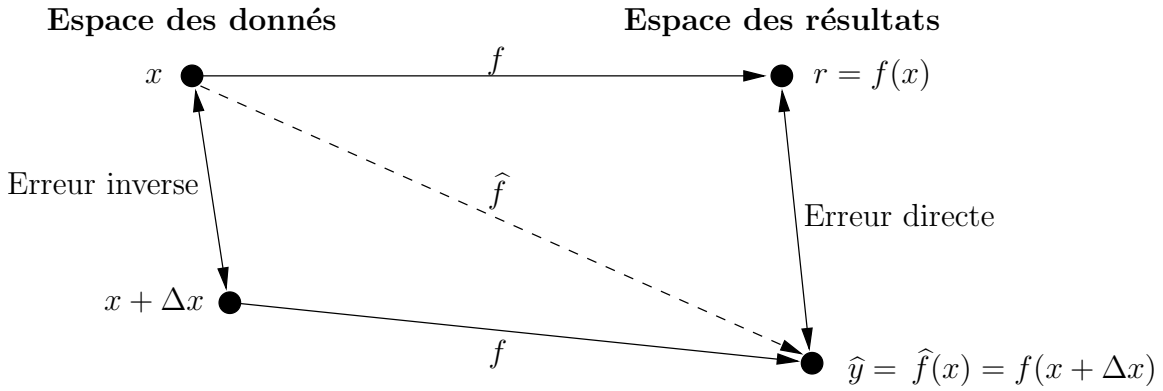


FIG. 2.1 – Erreur directe et erreur inverse lors du calcul de $\widehat{x} = \widehat{f}(x)$.

inverse-stable s'il produit toujours une solution dont l'erreur inverse est de l'ordre l'unité d'arrondi **u**.

2.2.3 Notion de conditionnement

Nous rappelons ici les notions de nombres de conditionnement absolu et relatif, dans le cas très simple où l'on s'intéresse à l'évaluation d'une fonction réelle de la variable réelle.

On considère une fonction réelle de la variable réelle f, que l'on suppose continûment dérivable au voisinage d'un réel x. On s'intéresse ici à l'effet sur $y = f(x)$ d'une perturbation absolue Δx, supposée infiniment petite, de la donnée x. En notant $\widehat{y} = f(x + \Delta x)$, on a au premier ordre,

$$\widehat{y} - y = f(x + \Delta x) - f(x) = f'(x)\Delta x + \mathcal{O}(|\Delta x|^2).$$

En notant $\mathrm{K}_a(f, x) = |f'(x)|$, la dérivée de f en x, on écrit

$$|\widehat{y} - y| = \mathrm{K}_a(f, x)|\Delta x| + \mathcal{O}(|\Delta x|^2).$$

Le facteur $\mathrm{K}_a(f, x)$ est le *nombre de conditionnement absolu* pour l'évaluation de f en la donnée x : ce facteur mesure l'effet d'une perturbation absolue de la donnée x sur le résultat $f(x)$.

Intéressons nous maintenant à l'effet sur $f(x)$ d'une perturbation relative $\Delta x/x$ de la donnée x. On a

$$\frac{\widehat{y} - y}{y} = \frac{xf'(x)}{f(x)}\frac{\Delta x}{x} + \mathcal{O}(|\Delta x|^2),$$

et en notant

$$\mathrm{K}_r(f, x) = \left|\frac{xf'(x)}{f(x)}\right|,$$

on obtient

$$\frac{|\widehat{y} - y|}{|y|} = \mathrm{K}_r(f, x)\frac{|\Delta x|}{|x|} + \mathcal{O}(|\Delta x|^2).$$

Le facteur $\mathrm{K}_r(f, x)$ traduit cette fois-ci l'effet d'une perturbation relative de la donnée x sur le résultat $f(x)$, et sera donc appelé *nombre de conditionnement relatif* pour l'évaluation de f en x.

La définition du conditionnement en termes de perturbations relatives est de loin la plus utilisée : nous parlerons donc plus simplement par la suite de *nombre de conditionnement*, et omettrons généralement l'adjectif *relatif*.

Comment s'interprète le nombre de conditionnement relatif ? Plus le conditionnement est grand, plus une petite perturbation relative des données peut entraîner une erreur relative importante sur la solution du problème. On dit qu'un problème est *mal conditionné* s'il admet un nombre de conditionnement élevé ; sinon on dit qu'il est *bien conditionné*. Cependant, la limite entre problèmes mal conditionnés et problèmes bien conditionnés dépendra à nouveau du contexte.

2.2.4 Précision du résultat d'un algorithme numérique

La précision de la solution calculée par un algorithme numérique à un problème donné dépend naturellement à la fois de ce problème et de la stabilité de l'algorithme utilisé.

La notion de conditionnement permet de faire le lien entre erreur directe et erreur inverse. En effet, l'erreur inverse modélise l'effet des erreurs générées par les calculs en précision finie comme des perturbations des données du problème. D'autre part, le conditionnement mesure l'effet de perturbations des données du problème sur sa solution exacte.

Lorsque l'erreur directe, l'erreur inverse et le conditionnement sont définis de façon relative, alors la précision du résultat calculé par un algorithme numérique vérifie l'estimation empirique suivante [39, p. 9] :

$$\text{précision} \lesssim \text{conditionnement} \times \text{erreur inverse}.$$

Il est à noter que la solution calculée en précision finie d'un problème mal conditionné peut présenter une précision arbitrairement mauvaise, même si l'erreur inverse associée à cette solution est « petite ».

En particulier, si l'on utilise un algorithme inverse-stable en arithmétique flottante, l'erreur inverse est par définition de l'ordre de l'unité d'arrondi $\mathbf{u}$. La précision de la solution calculée vérifie donc dans ce cas

$$\text{précision} \lesssim \text{conditionnement} \times \mathbf{u}.$$

Il est à noter que lorsque la conditionnement est supérieur à $\mathbf{u}^{-1}$, la relation précédente ne permet pas de garantir une précision du résultat inférieure à un. On en déduit une caractérisation généralement admise de la séparation entre problèmes bien conditionnés et problèmes mal conditionnés. On dira qu'un problème est bien conditionné à la précision de calcul $\mathbf{u}$ si son nombre de conditionnement est petit devant $\mathbf{u}^{-1}$. Au contraire, un problème dont le nombre de conditionnement est supérieur à $\mathbf{u}^{-1}$ sera dit mal conditionné.

2.3 Arithmétique flottante

Les nombres à virgule flottante, appelés plus simplement *nombres flottants*, constituent sur les calculateurs actuels, le principal mode de représentation des nombres réels. Bien entendu, l'*arithmétique flottante* n'est qu'une approximation de l'arithmétique réelle.

Dans cette section, nous passons en revue les propriétés essentielles de l'arithmétique flottante, et donnons les principales caractéristiques de la norme IEEE-754 [41] qui régit le comportement du calcul flottant sur de nombreux ordinateurs. Nous présentons également les notations que nous utiliserons pour l'analyse d'erreur en arithmétique flottante.

Précisons que cette section est à nouveau très largement inspirée de [39, chap. 2 et 3] et [50, chap. 1].

2.3.1 Nombres flottants

Un nombre flottant normalisé x est un rationnel de la forme

$$x = (-1)^s \times m \times b^e, \tag{2.2}$$

où $(-1)^s$ représente le signe de x avec $s \in \{0, 1\}$, m est la mantisse en base b, et e est l'exposant de x. La base b est un entier naturel supérieur ou égal à 2. L'exposant est un entier relatif e tel que

$$e_{min} \leq e \leq e_{\max}, \tag{2.3}$$

avec e_{min} et $e_{\max}$ deux entiers relatifs donnés. Soit t l'entier naturel désignant le nombre de chiffres que compte la mantisse m en base b. Celle-ci s'écrit alors

$$m = x_0 + x_1 b^{-1} + \ldots x_{t-1} b^{1-t}, \tag{2.4}$$

avec $x_0 \neq 0$ et $x_i \in \{0, 1, 2, \ldots, b-1\}$, pour $i = 0, \ldots, t-1$. La condition $x_0 \neq 0$ assure l'unicité de la représentation des nombres flottants normalisés, mais nous prive du nombre 0. On définit donc l'ensemble des flottants normalisés comme l'ensemble des valeurs définies par les relations (2.2), (2.3) et (2.4), union le singleton $\{0\}$.

On peut remarquer que la distance séparant deux flottants normalisés d'exposant e adjacents est b^{1-t+e}. Définissons la précision machine ϵ_m comme la distance séparant le flottant 1 du flottant normalisé suivant. Clairement, on a $\epsilon_m = b^{1-t}$.

D'autre part, tout nombre flottant normalisé x vérifie la relation suivante,

$$\lambda = b^{emin} \leq |x| \leq (1 - b^{-t}) b^{emax} = \sigma.$$

L'ensemble des flottants est généralement enrichi des nombres dénormalisés, qui permettent de représenter des nombres plus petits que λ en valeur absolue. Les flottants dénormalisés sont de la forme

$$x = (-1)^s \times m \times b^{e_{min}},$$

où le premier chiffre de la mantisse est nul,

$$m = x_1 b^{-1} + \ldots x_{t-1} b^{1-t},$$

avec $x_i \in \{0, 1, 2, \ldots, b-1\}$, pour $i = 1, \ldots, t-1$. Le plus petit flottant dénormalisé positif est $\lambda\epsilon_m = b^{e_{min}+1-t}$. Remarquons que les flottants dénormalisés sont régulièrement espacés, et que la distance séparant deux dénormalisés adjacents est également $\lambda\epsilon_m$.

Par la suite nous désignerons par $\mathbf{F}$ l'ensemble des nombres flottants, normalisés ou dénormalisés.

2.3.2 Notion d'arrondi et opérations flottantes

L'ensemble des nombres flottants $\mathbf{F}$ défini ci-dessus est utilisé pour approcher l'ensemble des réels $\mathbf{R}$. Pour cela il faut définir une application $\mathrm{fl} : \mathbf{R} \to \mathbf{F}$, utilisée pour associer à un réel sa représentation approchée sous la forme d'un flottant.

Définition 2.1. Soit $\mathrm{fl} : \mathbf{R} \to \mathbf{F}$ une fonction. On dit que fl est un arrondi de $\mathbf{R}$ vers $\mathbf{F}$ si les deux propriétés suivantes sont vérifiées :
- pour tout $x \in \mathbf{F}$, $\mathrm{fl}(x) = x$,
- pour tout $x, y \in \mathbf{R}$ tels que $x \leq y$, $\mathrm{fl}(x) \leq \mathrm{fl}(y)$.

On rencontre plusieurs types d'arrondi. Soient $x \in \mathbf{R}$ et $\mathrm{fl}(x)$ son arrondi.
- L'arrondi au plus près vérifie $|x - \mathrm{fl}(x)| = \min_{y \in \mathbf{F}} |x - y|$, avec bien entendu une règle du choix de $\mathrm{fl}(x)$ lorsque x est équidistant de deux flottants consécutifs.
- Les arrondis dirigés, sont
  - l'arrondi vers 0, tel que $|\mathrm{fl}(x)| \leq |x|$, et
  - les arrondis vers $+\infty$ et vers $-\infty$, vérifiant respectivement $x \leq \mathrm{fl}(x)$ et $\mathrm{fl}(x) \leq x$.

Ces quatre types d'arrondis sont ceux proposés par la norme IEEE-754.

Rappelons que pour tout flottant y normalisé, $\lambda \leq |y| \leq \sigma$. On considère $x \in \mathbf{R}$, que l'on souhaite arrondir dans $\mathbf{F}$. Deux cas particuliers peuvent se présenter.
- Si $|x| > \sigma$, on dit qu'il y a dépassement de capacité, ou overflow : le calcul pourra être interrompu, ou une valeur spéciale générée.
- Si $|x| < \lambda$, il y a dépassement de capacité inférieur, ou underflow : si le système flottant utilisé n'autorise pas les nombres dénormalisés, le résultat de l'arrondi pourra être 0 ou $\pm\lambda$. Si par contre le système flottant admet les nombres dénormalisés, alors $\mathrm{fl}(x)$ sera soit 0, soit le flottant dénormalisé adéquat. On dit alors que le dépassement de capacité inférieur est graduel (« gradual underflow »).

La manière dont ces cas de dépassement de capacité sont traités dans le cadre de la norme IEEE-754 sera décrit ci-après. Nous supposerons toujours par la suite que l'underflow est graduel.

Les opérations arithmétiques sur $\mathbf{F}$ sont définies à l'aide de la notion d'arrondi. Étant donnés $a, b \in \mathbf{F}$, ainsi qu'une opération arithmétique exacte $\circ \in \{+, -, \times, /\}$, l'arrondi du résultat exact $a \circ b$ sera considéré comme le résultat calculé en arithmétique flottante de cette opération. Par la suite, nous noterons souvent $\oplus$, $\ominus$, $\otimes$ et $\oslash$ les opérations flottantes, de manière à les distinguer des opérations exactes $+$, $-$, $\times$ et $/$.

Définissons ici clairement ce que l'on entend par dépassement de capacité inférieur, ou underflow, pour une opération arithmétique flottante. Cette définition est similaire à celle que l'on peut trouver dans [74, p.44].

Définition 2.2. Soient $a, b \in \mathbf{F}$, et $\circ \in \{+, -, \times, /\}$ une opération arithmétique exacte. On dit que l'opération flottante $\mathrm{fl}(a \circ b)$ provoque un dépassement de capacité inférieur, ou underflow, si le résultat exact $|a \circ b|$ est non nul, et vérifie $|a \circ b| < \lambda$.

Comme on suppose ici que l'underflow est graduel, le résultat d'une opération ayant provoqué un underflow sera soit 0, soit λ, soit le flottant dénormalisé adéquat. Par extension, on dira qu'un algorithme numérique s'exécute en l'absence d'underflow si aucune des opérations flottantes qu'il comporte ne provoque d'underflow.

D'autres définitions de l'underflow sont possibles. On peut en particulier considérer qu'il y a underflow lorsque $|\mathrm{fl}(a \circ b)| < \lambda$ (voir [39, p.38]). Néanmoins, nous adoptons la

Définition 2.2, car elle garantit le fait que le modèle standard s'applique (voir théorème 2.3 ci-dessous), et peut être testée à l'aide des drapeaux définis par la norme IEEE-754, et disponibles sur les unités flottantes actuelles [22, 41, 21, 32, 17, 43, 40]. Nous rappelons ci-après le modèle standard de l'arithmétique flottante, ainsi que son extension prenant en compte la possibilité d'underflow.

Définissons également formellement ce ce que l'on entend par arrondi fidèle. On définit pour cela le prédécesseur et le successeur dans $\mathbf{F}$ d'un réel r par

$$\text{pred}(r) = \max\{f \in \mathbf{F}/f < r\}, \quad \text{et} \quad \text{succ}(r) = \min\{f \in \mathbf{F}/r < f\}.$$

On dit que le flottant f est un arrondi fidèle du réel r si $\text{pred}(f) < r < \text{succ}(f)$. Notons que l'arrondi fidèle n'est pas un arrondi au sens de la définition 2.1 : en effet, la notion d'arrondi fidèle ne permet pas d'associer à tout réel un unique représentant dans $\mathbf{F}$.

2.3.3 Modélisation de l'arithmétique flottante

En l'absence d'underflow, l'erreur relative commise en effectuant une opération arithmétique flottante est majorée de manière relative par l'unité d'arrondi $\mathbf{u}$, qui dépend de la précision machine ϵ_m et du mode d'arrondi :

- $\mathbf{u} = \epsilon_m/2$ en arrondi au plus proche, et
- $\mathbf{u} = \epsilon_m$ en arrondi dirigé.

Le théorème suivant [39, chap.2] est classiquement utilisé.

Théorème 2.3 (Modèle standard de l'arithmétique flottante). *Soient a et b deux flottants, et $\circ \in \{+, -, \times, /\}$ une opération élémentaire. On suppose que l'opération $a \circ b$ ne provoque pas d'underflow, i.e., $\lambda \leq |a \circ b| \leq \sigma$. Alors,*

$$\text{fl}(a \circ b) = (a \circ b)(1 + \varepsilon_1) = (a \circ b)/(1 + \varepsilon_2), \quad \textit{avec} \quad |\varepsilon_1|, |\varepsilon_2| \leq \mathbf{u}. \tag{2.5}$$

Le modèle de l'arithmétique flottante décrit par le théorème 2.3 ignore la possibilité d'underflow. Pour prendre en compte une telle possibilité, on utilisera le théorème 2.4. Nous supposons ici que notre système de nombres flottants $\mathbf{F}$ admet les dénormalisés : l'underflow est graduel.

On définit l'unité d'underflow $\mathbf{v}$ comme étant le plus petit flottant positif dénormalisé [70, 86]. Notons que $\mathbf{v}$ est indépendant du mode d'arrondi utilisé, contrairement à l'unité d'arrondi $\mathbf{u}$. On a par définition $\mathbf{v} = \lambda\epsilon_m$, et dans le mode d'arrondi au plus proche $\mathbf{v} = 2\lambda\mathbf{u}$.

Si le résultat d'une addition provoque un underflow, alors on sait que le résultat de cette addition est exacte : le modèle (2.5) s'applique donc à toute addition. Par contre, si une multiplication produit un underflow, une perte de précision importante peut se produire, qui ne peut être majorée de façon relative comme dans (2.5) : il faut introduire une erreur absolue η, majorée par l'unité d'underflow $\mathbf{v}$ (voir [22, 70] et [39, p.56]).

Théorème 2.4 (Modèle de l'arithmétique flottante en présence d'underflow). *Soient a et b deux flottants. Si $\circ \in \{+, -\}$, alors*

$$\text{fl}(a \circ b) = (a \circ b)(1 + \varepsilon_1) = (a \circ b)/(1 + \varepsilon_2), \quad \textit{avec} \quad |\varepsilon_1|, |\varepsilon_2| \leq \mathbf{u}. \tag{2.6}$$

Si $\circ \in \{\times, /\}$, *alors*

$$\mathrm{fl}(a \circ b) = (a \circ b)(1+\varepsilon_1) + \eta_1 = (a \circ b)/(1+\varepsilon_2) + \eta_2,$$
$$\textit{avec} \quad |\varepsilon_1|, |\varepsilon_2| \leq \mathbf{u}, \quad |\eta_1|, |\eta_2| \leq \mathbf{v}, \quad \textit{et} \quad \varepsilon_1\eta_1 = \varepsilon_2\eta_2 = 0. \quad (2.7)$$

2.3.4 Norme IEEE-754

La norme IEEE-754, publiée en 1985 [41], définit une arithmétique flottante binaire. Elle est le résultat d'un groupe de travail du « IEEE Computer Society Computer Standards Committee ». La norme spécifie le format des nombres flottants, le résultat des opérations flottantes élémentaires et de leurs comparaisons, les modes d'arrondi ainsi que les conversions entre les différents formats arithmétiques. La racine carrée est incluse dans les opérations de base, mais la norme ne spécifie rien à propos des fonctions élémentaires telles exp et cos.

Deux principaux formats sont définis par la norme :

Type	Taille	Mantisse	Exposant			Intervalle
				e_{min}	e_{max}	
Simple	32 bits	23+1 bits	8 bits	-126	127	$\approx 10^{\pm 38}$
Double	64 bits	52+1 bits	11 bits	-1022	1023	$\approx 10^{\pm 308}$

Dans les deux formats, un bit est réservé pour le signe. Les nombres définis par la norme sont par défaut supposés normalisés, ce qui signifie que le premier bit de la mantisse est un 1. Ce bit ne sera donc pas stocké : on l'appel *bit implicite*, ou *bit caché*. Il est à noter que la norme prévoit également deux formats optionnels, le format simple étendu et le format double étendu, mais nous ne détaillons pas ici leurs spécifications.

Le standard spécifie que les opérations arithmétiques doivent donner un résultat comme si on calculait avec une précision infinie et que l'on arrondissait ensuite le résultat. La norme prévoit quatre types d'arrondi. L'arrondi par défaut est l'arrondi au plus près avec arrondi pair (zéro comme dernier bit si l'on se situe au milieu de deux nombres flottants). L'arrondi vers $-\infty$ et $+\infty$ est aussi supporté par la norme ; cela facilite l'implémentation de l'arithmétique d'intervalle par exemple. Le quatrième mode d'arrondi est l'arrondi vers zéro (appelé également la troncature).

Le tableau suivant donne les valeurs des constantes ϵ_m, λ, $\mathbf{u}$ et $\mathbf{v}$ pour la simple et la double précision IEEE-754, en fonction du mode d'arrondi courant.

Mode d'arrondi	Constante	simple précision	double précision
Indépendant du mode d'arrondi	Précision machine ϵ_m	2^{-23}	2^{-52}
	Plus petit normalisé λ	2^{-126}	2^{-1022}
	Unité d'underflow $\mathbf{v}$	2^{-149}	2^{-1074}
Arrondi au plus proche	Unité d'arrondi $\mathbf{u}$	2^{-24}	2^{-53}
Arrondis dirigés	Unité d'arrondi $\mathbf{u}$	2^{-23}	2^{-52}

La norme IEEE-754 fournit un système clos dans le sens où chaque opération produit un résultat. Elle définit pour ce faire des valeurs spéciales :

- les valeurs $+\infty$ et $-\infty$ représentent les infinis positifs et négatifs. Elles sont retournées comme résultat d'un overflow.
- les valeurs 0^+ et 0^- correspondent à l'inverse des valeurs infinies.

– la valeur NaN (Not a Number) correspond à des résultats indéterminés.

Cinq anomalies (« exceptions ») sont également définies : chaque anomalies devra être signalée grâce à la modification d'un indicateur d'état (« status flag »). L'utilisateur doit également avoir à sa disposition un piège (« trap ») pour chaque anomalie, qu'il peut choisir de désactiver. Lors de l'occurrence d'une anomalie, celle-ci sera provoquera donc nécessairement la modification de l'indicateur d'état correspondant, et éventuellement le déroutement vers un piège.

Notons que la norme préconise le déroulement ininterrompu du programme, c'est à dire la continuation sans piège : dans ce cas, une valeur spéciale est renvoyée. Les cinq anomalies définies sont les suivantes.

1. *Opération invalide* : l'opération mathématique est invalide ou son résultat est indéterminé.
2. *Division par zéro* : le diviseur est nul alors que le dividende est non nul.
3. *Dépassement de capacité vers l'infini* : le résultat d'un calcul excède en valeur absolue la plus grande valeur représentable.
4. *Dépassement de capacité inférieur* : le résultat de l'opération est non nul, mais strictement plus petit en valeur absolue que le plus petit flottant normalisé.
5. *Indicateur d'inexactitude* : le résultat arrondi est différent du résultat exact.

Le tableau ci-dessous donne des exemples d'opérations provoquant une anomalie, en indiquant quelles sont les valeurs pouvant être renvoyées.

Type d'exception	Exemple	Résultats
Opérations invalide	$0/0, 0\times\infty, \sqrt{-1}$	NaN (Not a Number)
Division par zéro	nombre fini$/0$	$\pm\infty$
Dépassement de capacité vers l'infini		$\pm\infty$
Dépassement de capacité inférieur		nombres dénormalisés
Indicateur d'inexactitude	quand $\mathrm{fl}(x\circ y)\neq x\circ y$	arrondi correct

2.3.5 Notations pour l'analyse d'erreur

Les notations que nous utilisons sont décrites dans [39, chap. 3]. Nous rappelons néanmoins ci-dessous celles qui seront le plus souvent utilisées dans nos analyses d'erreur.

Pour prendre en compte les termes de la forme $\prod(1+\varepsilon_i)^{\pm 1}$, nous utiliserons les notations θ_k et γ_k, définies dans le lemme suivant.

Lemme 2.5. *Si* $|\varepsilon_i|\leq \mathbf{u}$ *pour* $i=1:k$ *et* $k\mathbf{u}<1$, *alors*

$$\prod_{i=1}^{k}(1+\varepsilon_i)^{\pm 1}=1+\theta_k,\quad \textit{avec}\quad |\theta_k|\leq\frac{k\mathbf{u}}{1-k\mathbf{u}}=:\gamma_k.$$

Parfois, pour éviter une profusion de facteurs $(1+\theta_k)$, nous utiliserons également le compteur d'erreur $\langle k\rangle$, pour désigner un produit de la forme

$$\langle k\rangle=\prod_{i=1}^{k}(1+\varepsilon_i)^{\pm 1},\quad \text{avec}\quad |\varepsilon_i|\leq\mathbf{u}.$$

On a donc $\langle k \rangle = (1 + \theta_k)$, avec $|\theta_k| \leq \gamma_k$. Les θ_k et les compteurs d'erreurs $\langle k \rangle$ sont manipulés à l'aide des deux égalités suivantes :

$$(1 + \theta_i)(1 + \theta_j) = 1 + \theta_{i+j}, \quad \text{et} \quad \langle i \rangle \langle j \rangle = \langle i + j \rangle.$$

En utilisant ces notations, on suppose toujours implicitement $k\mathbf{u} < 1$.

Il est important de noter les relation suivantes à propos des γ_k, démontrées dans [39, chap. 3] :

$$\begin{aligned} k\mathbf{u} &\leq \gamma_k, \\ i\gamma_k &\leq \gamma_{ik}, \\ \gamma_k + \gamma_j + \gamma_k\gamma_j &\leq \gamma_{k+j}, \\ (1 + \mathbf{u})^k &\leq 1 + \gamma_k, \end{aligned}$$

et

$$(1 + \mathbf{u})\gamma_k \leq (1 - \mathbf{u})^{-1}\gamma_k \leq \gamma_{k+1}.$$

On définit $\widehat{\gamma}_k \in \mathbf{F}$, l'évaluation en arithmétique flottante de γ_k, par

$$\widehat{\gamma}_k = (k \otimes \mathbf{u}) \oslash (1 - k\mathbf{u}).$$

Dans la relation précédente, comme $k\mathbf{u} < 1$, il est important de noter que $1 - k\mathbf{u} \in \mathbf{F}$. Cela signifie qu'une seule erreur d'arrondi sera commise lors du calcul de $\widehat{\gamma}_k$, d'où les inégalités suivantes, qui se déduisent facilement du modèle standard (2.5) :

$$\gamma_k \leq (1 + \mathbf{u})\widehat{\gamma}_k \leq (1 - \mathbf{u})^{-1}\widehat{\gamma}_k.$$

La borne suivante provient d'une application directe du modèle standard (2.5) — voir [71] pour une preuve de cette inégalité. Pour $x \in \mathbf{F}$ tel que $|x| \geq \lambda$, et $k \in \mathbf{N}$ tel que $(k + 1)\mathbf{u} < 1$, on a

$$(1 + \mathbf{u})^k |x| \leq \mathrm{fl}\left(\frac{|x|}{1 - (k + 1)\mathbf{u}}\right). \tag{2.8}$$

Cette inégalité sera utilisée notamment pour obtenir des bornes d'erreurs *a posteriori*.

2.4 Exemple du schéma de Horner

À titre d'exemple, montrons comment procéder à l'analyse d'erreur du schéma de Horner, avec les notations vues dans les sections précédentes. On considère un polynôme p de degré n, dont les coefficients sont supposés flottants :

$$p(x) = \sum_{i=0}^{n} a_i x^i.$$

Étant donné un nombre flottant x, on désigne par $\mathsf{Horner}(p, x)$ le résultat de l'évaluation de $p(x)$ en arithmétique flottante à l'aide du schéma de Horner rappelé ci-dessous. On suppose ici que l'on travaille en arrondi au plus proche, et qu'aucun dépassement de capacité (inférieur ou supérieur) n'intervient au cours des calculs.

Algorithme 2.6. Schéma de Horner

function $\widehat{r}_0 = \mathsf{Horner}(p, x)$
  $\widehat{r}_n = a_n$
  for $i = n-1 : -1 : 0$
    $\widehat{r}_i = \widehat{r}_{i+1} \otimes x \oplus a_i$
  end

Le nombre de conditionnement pour l'évaluation du polynôme $p(x) = \sum_{i=0}^{n} a_i x^i$, en un point x donné, est défini classiquement par [20]

$$\mathrm{cond}(p, x) := \frac{\widetilde{p}(x)}{|p(x)|}, \quad \text{avec} \quad \widetilde{p}(x) = \sum_{i=0}^{n} |a_i||x|^i. \tag{2.9}$$

On notera que le nombre de conditionnement $\mathrm{cond}(p, x)$ peut être arbitrairement large, par exemple au voisinage des racines de p.

Montrons comment obtenir un résultat de nature inverse pour le résultat $\mathsf{Horner}(p, x)$ de l'évaluation de $p(x)$ par le schéma de Horner. Pour ce faire, utilisons les compteurs d'erreur $\langle k \rangle$ définis à la section précédente. En utilisant les notations de l'algorithme 2.6, on a

$$\begin{aligned}
\widehat{r}_n &= a_n \\
\widehat{r}_{n-1} &= \widehat{r}_n \otimes x \oplus a_{n-1} = \langle 1 \rangle \big(\langle 1 \rangle \widehat{r}_n x + a_{n-1} \big) \\
&= \langle 2 \rangle a_n x + \langle 1 \rangle a_{n-1}, \\
\widehat{r}_{n-2} &= \widehat{r}_{n-1} \otimes x \oplus a_{n-2} = \langle 1 \rangle \big(\langle 1 \rangle \widehat{r}_{n-1} x + a_{n-2} \big) \\
&= \langle 4 \rangle a_n x^2 + \langle 3 \rangle a_{n-1} x + \langle 1 \rangle a_{n-2}.
\end{aligned}$$

Alors, on peut montrer par induction que

$$\widehat{r}_0 = \langle 2n \rangle a_n x^n + \sum_{i=0}^{n-1} \langle 2i+1 \rangle a_i x^i.$$

Chacun des $\langle k \rangle$ dans l'égalité précédente représente un facteur $(1 + \theta_k)$, avec $|\theta_k| \leq \gamma_k$. Comme de plus $\widehat{r}_0 = \mathsf{Horner}(p, x)$, on obtient

$$\mathsf{Horner}(p, x) = (1 + \theta_{2n}) a_n x^n + \sum_{i=0}^{n-1} \big(1 + \theta_{2i+1}\big) a_i x^i,$$

où chacun des θ_k vérifie $|\theta_k| \leq \gamma_k$. On peut simplifier légèrement ce résultat de nature inverse en écrivant

$$\mathsf{Horner}(p, x) = \sum_{i=0}^{n} (1 + \delta_i) a_i x^i, \quad \text{avec} \quad |\delta_i| \leq \gamma_{2n}.$$

La valeur calculée $\mathsf{Horner}(p, x)$ est donc l'évaluation exacte d'un polynôme dont les coefficients sont ceux de p, perturbés de perturbations relatives δ_i. L'erreur relative inverse, mesurée comme étant la plus grande des perturbations relative des coefficients de p, est

donc majorée par $\gamma_{2n} \approx 2n\mathbf{u}$. Puisque cette erreur relative inverse est de l'ordre de grandeur de l'unité d'arrondi $\mathbf{u}$, quelque que soient les coefficients de p et quel que soit x, l'algorithme de Horner est un algorithme inverse-stable [20, p. 16],[39, p. 95].

En utilisant le résultat inverse précédent, on obtient facilement une borne sur l'erreur directe entachant $\mathsf{Horner}(p, x)$. En effet, on a

$$|\mathsf{Horner}(p, x) - p(x)| = \left| \sum_{i=0}^{n} (1 + \delta_i) a_i x^i - \sum_{i=0}^{n} a_i x^i \right| = \left| \sum_{i=0}^{n} \delta_i a_i x^i \right|,$$

avec $|\delta_i| \leq \gamma_{2n}$. On obtient donc la borne suivante sur l'erreur directe absolue entachant $\mathsf{Horner}(p, x)$:

$$|\mathsf{Horner}(p, x) - p(x)| \leq \gamma_{2n} \sum_{i=0}^{n} |a_i||x^i|.$$

La précision du résultat de l'évaluation calculée par l'algorithme 2.6 est donc majorée en fonction du nombre de conditionnement $\mathrm{cond}(p, x)$ selon l'inégalité suivante,

$$\frac{|p(x) - \mathsf{Horner}(p, x)|}{|p(x)|} \leq \gamma_{2n} \, \mathrm{cond}(p, x). \tag{2.10}$$

En particulier, lorsque $\mathrm{cond}(p, x) > \gamma_{2n}^{-1}$, on ne peut garantir une erreur relative inférieure à un. C'est le cas en particulier au voisinage des racines de p.

2.5 Améliorer la précision du résultat

Supposons que l'on travaille sur une architecture dont l'arithmétique flottante est conforme à la norme IEEE-754. Que faire lorsque la double précision IEEE-754 n'est pas suffisante pour garantir la précision du résultat ? Citons à ce sujet Bailey [6] :

> For most scientific applications, [IEEE 64-bit floating-point arithmetic] is more than sufficient, and for some applications, such as routine processing of experimental data, even the 32-bit standard often provides sufficient accuracy.
>
> However, for a rapidly expanding body of applications, 64-bit IEEE arithmetic is no longer sufficient. These range from some interesting new mathematical computations to large-scale physical simulations performed on highly parallel supercomputers. In these applications, portions of the code typically involve numerically sensitive calculations, which produce results of questionable accuracy using conventional arithmetic.

Bailey insiste ensuite sur la demande croissante, notamment pour les simulations physiques, quant à la possibilité de disposer d'une précision supérieure à la double précision IEEE-754. Cette demande est également soulignée par De Dinechin et Villard dans [18].

La réponse classique pour améliorer la précision du résultat est donc d'augmenter la précision des calculs. Dans la suite de cette section, nous passons en revue différentes solutions génériques permettant d'augmenter la précision de travail. Nous présenterons également le principe général des algorithmes compensés, en faisant le lien avec la méthode CENA développée par Langlois [49].

2.5.1 Méthodes génériques pour augmenter la précision de calcul

Lorsque la double précision IEEE-754 n'est pas suffisante pour garantir une précision donnée, diverses solutions classiques existent pour augmenter la précision des calculs.

Tout d'abord, il est possible d'utiliser une arithmétique exacte, permettant de ne jamais perdre de précision, en utilisant par exemple une représentation rationnelle ou modulaire des nombres. Citons par exemple la bibliothèque GMP [1] qui permet d'effectuer des calculs sur des entiers arbitrairement longs, ou sur l'ensemble des rationnels. Cependant, l'utilisation du calcul exact reste peu répandue dans le domaine du calcul scientifique : ce type d'arithmétique reste en effet très coûteux en termes de temps de calcul face aux implantations matérielles de l'arithmétique flottante. De plus, la réalisation de calculs exacts n'est pas toujours possible : le résultat d'un calcul n'est pas nécessairement un nombre rationnel, ni même un nombre algébrique.

Notons qu'il est possible, dans certains cas, d'utiliser la double précision étendue, dont l'implantation matérielle est encouragée par la norme IEEE-754. Il convient cependant de noter que cette double précision étendue n'est pas disponible sur tous les processeurs modernes — elle n'est pas implantée par exemple sur l'architecture PowerPC d'IBM. L'utilisation de la double précision étendue ne garantit donc pas la portabilité du logiciel.

Les arithmétiques multiprécisions permettent à l'utilisateur de choisir une précision de calcul arbitrairement élevée. L'un des exemples les plus connus de bibliothèque de calcul flottant multiprécision est la bibliothèque MP de Brent [13]. Un exemple plus récent est la bibliothèque ARPREC [7], faisant suite à MPFUN90 [5, 4] de Bailey. Citons également la bibliothèque MPFR [31] développée en France par l'INRIA. MPFR est en partie basée sur GMP, et propose les quatre modes d'arrondis pour les opérations élémentaires, ainsi que les principales fonctions élémentaires avec arrondi correct.

Les expansions de longueur fixe, telles les double-doubles et les quad-doubles sont des solutions possibles pour simuler respectivement deux fois ($\mathbf{u}^2$) et quatre fois ($\mathbf{u}^4$) la double précision IEEE-754 ($\mathbf{u}$). L'arithmétique flottante est alors redéfinie sur des quantités représentées par la somme non évaluée de deux ou quatre flottants en double précision. Le format double-double est utilisable de façon portable au travers des bibliothèques proposées par Bailey [4, 34] ou Briggs [14]. La bibliothèque QD [34] de Bailey donne à la fois accès aux formats double-double et quad-double.

L'arithmétique double-double constitue aujourd'hui une référence dans le domaine du calcul scientifique, pour simuler une précision de travail équivalente à deux fois la double précision IEEE-754 de façon efficace. Les double-doubles sont notamment utilisés au sein de la bibliothèque *Extended and Mixed precision BLAS* [56, 57].

Par la suite, nous utiliserons les bibliothèques QD [34], MPFR [31] et XBLAS [56] pour comparer leurs performances à celles des algorithmes compensés.

2.5.2 Algorithmes compensés

Dans le cadre de la norme IEEE-754 et dans le mode d'arrondi au plus proche, si a et b sont deux nombres flottants, il est bien connu que l'erreur d'arrondi commise en calculant l'addition $\mathrm{fl}(a+b)$ est elle-même un nombre flottant [19]. On peut alors écrire

$$x + \delta = a + b, \quad \text{avec} \quad x = \mathrm{fl}(a+b) \quad \text{et } \delta \text{ un flottant.} \tag{2.11}$$

Le terme δ dans l'égalité précédente est l'erreur d'arrondi générée lors du calcul de l'addition flottante $\mathrm{fl}(a+b)$. Ce résultat se généralise au cas de la multiplication et, sous une forme

différente, au cas de la division, comme nous le verrons plus loin.

D'autre part, on connaît des algorithmes qui, en utilisant uniquement la précision de travail et des opérations arithmétiques élémentaires, permettent de calculer δ et y vérifiant (2.11) en fonction de a et de b. L'algorithme de Knuth [47], que nous appellerons par la suite TwoSum, permet notamment ce calcul.

L'addition $a+b$ peut donc être transformée exactement à l'aide de TwoSum en la somme non évaluée $x + \delta$. Une telle transformation est appelée « error-free transformation » par Ogita, Rump et Oishi dans [70], ce nous traduirons ici par « transformation exacte ».

Le fait de calculer exactement l'erreur d'arrondi y générée par l'addition flottante $a \oplus b$ permet éventuellement, d'en corriger son effet sur les opérations arithmétiques suivantes. Afin d'expliquer plus précisément le principe général des algorithmes compensés, considérons une fonction $f : \mathbf{R}^n \to \mathbf{R}$. Soit $x = (x_1, \ldots, x_n) \in \mathbf{F}^n$ un jeu de données. Le réel $f(x)$ est approché à l'aide d'un algorithme $\widehat{f}$ utilisant l'arithmétique flottante, et faisant intervenir les variables intermédiaires flottantes $\widehat{x}_{n+1}, \ldots, \widehat{x}_N$. En outre, on suppose que $\widehat{f}(x) = \widehat{x}_N$, et que chacun des $\widehat{x}_k$, pour $k > n$, est le résultat d'une opération élémentaire, $+, -, \times, /$ effectuée en arithmétique flottante : on note δ_k l'erreur d'arrondi associée au calcul de $\widehat{x}_k$, et $\delta = (\delta_{n+1}, \ldots, \delta_N)$. Comme nous l'avons déjà dit, les erreurs d'arrondi δ_k peuvent soit être calculées exactement, soit être approchées très précisément, à l'aide des transformations exactes.

Insistons sur le fait que, dans le cas général, $\widehat{f}(x)$ est entaché d'une erreur directe non nulle. Définissons donc le *terme correctif exact* pour $\widehat{f}(x)$ par

$$c := f(x) - \widehat{f}(x).$$

Supposons que c puisse être exprimé formellement en fonction de $x = (x_1, \ldots, x_n) \in \mathbf{F}^n$ et des erreurs d'arrondi $\delta = (\delta_{n+1}, \ldots, \delta_N)$. S'il était possible de calculer exactement c, alors nous pourrions calculer très précisément $f(x)$, puisque par définition de c, on a

$$f(x) = \widehat{x}_N + c.$$

Or en général, le terme correctif exact ne pourra pas être calculé exactement, mais seulement approché à l'aide d'un algorithme en arithmétique flottante : notons $\widehat{c}$ ce *terme correctif approché*. On définit alors le *résultat compensé* $\overline{r} \in \mathbf{F}$ par

$$\overline{x}_N = \widehat{x}_N \oplus \widehat{c}.$$

Puisque $f(x) = \widehat{x}_N + c$, et que $\widehat{c}$ est une valeur approchée c, on peut légitimement penser que le résultat compensé $\overline{x}_N$ est une approximation plus précise de $f(x)$ que $\widehat{x}_N$. Néanmoins, la véracité de cette affirmation dépend bien entendu de la manière dont est calculé le terme correctif approché $\widehat{c}$. De plus, il convient de prouver au cas par cas le fait que le résultat compensé est effectivement plus précis que le résultat initial. Le principe du calcul d'un résultat compensé sera bien entendu largement illustré dans les chapitres suivants.

Citons également la méthode CENA, développée par Langlois [49]. Le principe de la méthode CENA est d'utiliser un approximant d'ordre un pour le calcul du terme correctif approché. On a en effet

$$c = f(x) - \widehat{f}(x) = \sum_{k=n+1}^{N} \frac{\partial \widehat{f}}{\partial \delta_k}(X, \delta) \cdot \delta_k + E_L, \tag{2.12}$$

ou E_L désigne l'erreur de linéarisation. Les dérivées partielles de $\widehat{f}$ en (X, δ) sont approchées par différentiation automatique [29], et les erreurs d'arrondi δ_k sont calculées, ou approchées précisément, à l'aide des transformations arithmétiques exactes pour les opérations élémentaires. Le terme correctif approché $\widehat{c}$ est calculé à la précision courante d'après la relation (2.12), en supposant que l'erreur de linéarisation est nulle.

Dans [49], Langlois isole une classe d'algorithmes numériques pour laquelle l'erreur de linéarisation E_L est effectivement nulle, et appelle ces algorithmes *algorithmes linéaires*. Parmi ces algorithmes linéaires, on trouve l'algorithme de sommation récursive classique, le calcul de produits scalaires, le schéma de Horner, ainsi que la résolution de systèmes triangulaires. Dans le cas des systèmes triangulaires par exemple, Langlois observe expérimentalement dans [49] que la solution compensée calculée à l'aide de la méthode CENA est aussi précise que celle calculée par l'algorithme de substitution classique exécuté en précision doublée.

2.6 Conclusion

Dans ce chapitre, nous avons présenté les notions de bases pour l'analyse d'erreur en précision finie, en introduisant les notions d'erreur directe, d'erreur inverse et de nombre de conditionnement. L'erreur directe est la distance séparant la solution exacte d'un problème d'une solution approchée à ce problème, calculée par un algorithme donné. La notion d'erreur inverse permet de mettre en évidence la stabilité, ou au contraire l'instabilité d'un algorithme. Le conditionnement quant à lui régit le lien entre erreur directe et erreur inverse, en mesurant la sensibilité de la solution à des perturbations sur les données.

Nous avons également présenté les notions de base quant au calcul en arithmétique flottante, notamment au travers de son modèle standard. Les principales caractéristiques de la norme IEEE-754 ont été décrites.

En outre, nous avons rappelé l'analyse d'erreur classique du schéma de Horner. Comme nous l'avons vu, le schéma de Horner est un algorithme inverse stable. Néanmoins, la précision du résultat calculé par le schéma de Horner peut être arbitrairement mauvaise, en particulier lorsque le polynôme considéré est mal conditionné. Cela justifie l'étude au cours des chapitres suivants du schéma de Horner compensé, permettant d'accroître la précision de l'évaluation polynomiale.

Chapitre 3

Transformations arithmétiques exactes

Plan du chapitre : En Section 3.2, nous détaillons les transformations exactes que nous avons utilisées pour chacune des opérations arithmétiques élémentaires. Nous présenterons ensuite deux applications connues de ces transformations exactes. En Section 3.3 nous rappellerons en effet les résultats obtenus par Ogita, Rump et Oishi à propos de la sommation et du produit scalaire compensés, puis nous donnerons quelques détails sur les arithmétiques double-double et quad-double en Section 3.4

Hypothèse : Les résultats présentés dans la suite de cette thèse supposent l'utilisation de l'arithmétique binaire IEEE-754, en arrondi au plus proche, avec underflow graduel [41].

3.1 Introduction

Soient a et b deux flottants, et soit $\circ = +, -$ ou $\times$ une opération arithmétique. On considère $x = \mathrm{fl}(a \circ b)$ le résultat de l'opération flottante calculée dans le mode d'arrondi au plus proche. En l'absence d'underflow et d'overflow, il est connu que l'*erreur d'arrondi élémentaire* $y = (a \circ b) - \mathrm{fl}(a \circ b)$ est elle-même un flottant [9, 11], et l'on peut écrire

$$a \circ b = x + y, \quad \text{avec} \quad x = \mathrm{fl}(a \circ b) \in \mathbf{F}, \quad \text{et} \quad y \in \mathbf{F}. \tag{3.1}$$

D'autre part, on connaît des algorithmes qui, en utilisant uniquement la précision de travail et des opérations arithmétiques élémentaires, permettent de calculer x et y vérifiant (3.1) en fonction des entrées a et b. L'opération $a \circ b$ se trouve ainsi transformée exactement en la somme non évaluée des deux flottants x et y. Un algorithme permettant une telle transformation est appelé *error-free transformation* par Ogita, Rump et Oishi dans [70], ce que nous traduisons par *transformation exacte*.

Dans ce chapitre, nous passons en revue les transformations exactes que nous avons utilisées pour chacune des opérations élémentaires $+$, $-$ et $\times$. Nous supposons, comme dans la suite de cette thèse, que l'on travaille dans le mode d'arrondi au plus proche. Cette hypothèse assure en effet que les erreurs d'arrondi générées par les opérations $+$, $-$ et $\times$ sont représentables dans $\mathbf{F}$. Comme nous le verrons, elle garantie également des résultats similaires dans le cas de la division et du Fused-Mutiply-and-Add (FMA).

Ces transformations exactes constituent les briques de base pour la conception d'algorithmes compensés. À titre d'exemple d'application, et parce qu'ils seront utilisés dans

les chapitres suivants (notamment les chapitres 8 et 9), nous reproduisons dans cette section quelques résultats présents dans [70] à propos de la sommation et du produit scalaire compensés.

Nous définirons également le format double-double ainsi que les opérations arithmétiques associées. Rappelons que ce format est notamment utilisé pour simuler une précision de travail équivalente à deux fois la double précision IEEE-754 au sein de la bibliothèque *Extended and Mixed precision BLAS* [56, 57]. Cela nous permettra notamment de formuler le schéma de Horner effectué en arithmétique double-double, qui sera utilisé par la suite dans nos tests de performances. Nous définirons également plus brièvement le format quad-double.

3.2 Transformations exactes des opérations élémentaires

Passons en revue les transformations exactes connues pour les opérations élémentaires qui seront utilisées dans les chapitres suivants pour la description des algorithmes compensés. Nous nous inspirons des références [9, 11, 70].

Rappelons que nous utilisons l'arithmétique flottante IEEE-754 en arrondi au plus proche. Nous supposons que l'underflow est graduel, mais nous ne prendrons pas en compte les cas d'overflow, qui sont facilement détectables [41]. Notons que des hypothèses plus fines pour assurer le fait que les erreurs d'arrondi soient représentables dans $\mathbf{F}$ sont décrites dans [11]. Néanmoins, nous ne reproduisons pas ici les résultats de cet article, car nous ne les avons pas utilisés dans nos travaux.

3.2.1 Transformation exacte de l'addition

Étant donnés deux flottants a et b tels que $|a| \geq |b|$, l'algorithme $\mathsf{FastTwoSum}$ permet de calculer à la fois $x = a \oplus b$ et l'erreur d'arrondi élémentaire entachant ce résultat.

Algorithme 3.1. Transformation exacte de la somme de deux nombres flottants ordonnés.

function $[x, y] = \mathsf{FastTwoSum}(a, b)$
  $x = a \oplus b$
  $y = (a \ominus x) \oplus b$

Le résultat suivant est dû à Dekker [19], qui a montré en 1971 que $\mathsf{FastTwoSum}$ calcule effectivement l'erreur d'arrondi exacte lorsque les entrées vérifient $|a| \geq |b|$.

Théorème 3.2 ([19])**.** *Soient $a, b \in \mathbf{F}$ tels que $|a| \geq |b|$, et soient $x, y \in \mathbf{F}$ tels que $[x, y] = \mathsf{FastTwoSum}(a, b)$. On suppose que le mode d'arrondi courant est l'arrondi au plus proche. Alors, même en présence d'underflow,*

$$a + b = x + y, \quad x = a \oplus b, \quad |y| \leq \mathbf{u}|x|, \quad \textit{et} \quad |y| \leq \mathbf{u}|a + b|. \tag{3.2}$$

Remarque. Précisons que le théorème 3.2 s'applique également en base 3, mais pas dans des bases plus grandes [19]. De plus, la condition $|a| \geq |b|$ peut être remplacée par l'hypothèse moins restrictive $e_a \geq e_b$ [19], en notant e_z l'exposant d'un flottant z.

Si on ne dispose d'aucune information sur la manière dont a et b se comparent, il faut pour utiliser FastTwoSum introduire un branchement conditionnel, comme dans l'algorithme TwoSumCond ci-dessous.

Algorithme 3.3. Transformation exacte de la somme de deux nombres flottants.

function $[x, y] = \mathsf{TwoSumCond}(a, b)$
  if $|a| \geq |b|$
    $[x, y] = \mathsf{FastTwoSum}(a, b)$
  else
    $[x, y] = \mathsf{FastTwoSum}(b, a)$
  end

Il existe des architectures sur lesquelles il est préférable, du point de vue des performances, d'utiliser une autre variante de l'algorithme FastTwoSum. C'est le cas par exemple de l'architecture Itanium d'Intel, dont le jeu d'instructions comprend les instructions famin et famax [16]. Étant donnés deux flottants a et b tels que $|a| \geq |b|$, $\mathsf{famax}(a, b)$ retourne a et $\mathsf{famin}(a, b)$ retourne b. On peut donc utiliser sur l'architecture Itanium l'algorithme suivant qui présente l'avantage de ne comporter aucun branchement [16].

Algorithme 3.4. Transformation exacte de la somme de deux nombres flottants.

function $[x, y] = \mathsf{TwoSumMinMax}(a, b)$
  $max = \mathsf{famax}(a, b)$
  $min = \mathsf{famin}(b, a)$
  $[x, y] = \mathsf{FastTwoSum}(max, min)$

L'algorithme TwoSum suivant effectue le même travail, pour des entrées a et b quelconques sans utiliser de branchement. Cet algorithme est dû à Knuth en 1969 (voir [47]).

Algorithme 3.5. Transformation exacte de la somme de deux nombres flottants.

function $[x, y] = \mathsf{TwoSum}(a, b)$
  $x = a \oplus b$
  $z = x \ominus a$
  $y = (a \ominus (x \ominus z)) \oplus (b \ominus z)$

Remarque. Knuth démontre que l'algorithme 3.5 calcule effectivement l'erreur d'arrondi entachant l'addition $a \oplus b$ pour une arithmétique flottante en base $b \geq 2$ quelconque, mais en l'absence de dépassement de capacité. Boldo et Daumas démontrent que l'algorithme 3.5 est valide en base 2 avec underflow graduel [11].

L'algorithme TwoSum nécessite 6 opérations flottantes, alors que TwoSumCond n'en demande que 3. Cependant TwoSum est souvent privilégié en pratique, car il n'utilise aucun branchement conditionnel. Sur les processeurs superscalaires actuels, comme le Pentium d'Intel ou le Power d'IBM, de tels branchements conditionnels peuvent se révéler très coûteux en pratique, même s'ils permettent d'éviter quelques instructions flottantes. C'est pourquoi sauf mention contraire, nous utiliserons l'algorithme TwoSum par la suite.

Les algorithmes TwoSum, TwoSumCond, et TwoSumMinMax sont équivalents du point de vu numérique : étant données deux entrées a et b, ils produisent tous comme résultat la même paire x, y. Nous ne faisons donc référence qu'à TwoSum dans le théorème suivant.

Théorème 3.6. *Soient $a, b \in \mathbf{F}$ et $x, y \in \mathbf{F}$ tels que $[x, y] = \mathsf{TwoSum}(a, b)$. On suppose que le mode d'arrondi courant est l'arrondi au plus proche. Alors, même en présence d'underflow,*

$$a + b = x + y, \quad x = a \oplus b, \quad |y| \leq \mathbf{u}|x|, \quad et \quad |y| \leq \mathbf{u}|a + b|. \tag{3.3}$$

Indiquons que dans le cadre de l'arithmétique IEEE-754, il est nécessaire de supposer que le mode d'arrondi d'arrondi est l'arrondi au plus proche : dans [11] des contres exemples sont fournis, montrant que l'erreur d'arrondi sur l'addition n'est pas nécessairement représentable dans les autres modes d'arrondi définis par la norme.

3.2.2 Transformation exacte de la multiplication

Par la suite, nous utiliserons généralement l'algorithme de Dekker et Veltkamp (voir [19]) pour la transformation sans erreur de la multiplication. Cette méthode nous impose de travailler en arrondi au plus proche. D'autres algorithmes existent cependant pour les autres modes d'arrondis prévus par la norme : Priest décrit notamment dans [82] des transformations sans erreur pour la multiplication dans le cadre d'une arithmétique fidèle.

Pour utiliser la méthode de Dekker et Veltkamp, il faut au préalable découper les flottants en entrées en deux parties. Soient p le nombre de bits de la mantisse à la précision courante et $s = \lceil p/2 \rceil$. L'algorithme Split de Dekker, permet de découper un flottant a en deux flottants x et y.

Algorithme 3.7. Découpe d'un nombre flottant en deux parties.

function $[x, y] = \mathsf{Split}(a)$
  % $C = 2^s + 1$ est une constante pré-calculée
  $z = C \otimes a$
  $x = z \ominus (z \ominus a)$
  $y = a \ominus x$

Les propriétés de l'algorithme Split sont précisées dans le théorème 3.8. Rappelons que si $x, y \in \mathbf{F}$ sont tels que $|y| \leq |x|$, ont dit que x et y ne se chevauchent pas si le bit non nul le moins significatif de x est d'un poids strictement supérieur à celui du bit non nul le plus significatif de y.

Théorème 3.8 ([88])**.** *Soient p le nombre de bits de la mantisse à la précision courante et $s = \lceil p/2 \rceil$. Soit $a \in \mathbf{F}$ et soient $x, y \in \mathbf{F}$ tels que $[x, y] = \mathsf{Split}(a)$. Alors, en l'absence d'underflow, x compte au plus $p - s$ bits non nuls, y compte au plus $s - 1$ bits non nuls et l'on a*

$$a = x + y, \quad x \text{ et } y \text{ ne se chevauchent pas}, \quad et \quad |y| < |x|. \tag{3.4}$$

Il est intéressant de noter que x et y comptent ensemble au plus $p-1$ bits non nuls, que p soit pair ou impair. Ceci est dû au fait qu'en base 2, et en arrondi au plus proche, le bit de signe de y permet de compenser ce bit manquant. Considérons par exemple, en travaillant avec une mantisse à 7 bits, $a = 1101001$: le flottant a sera découpé en $x = 1110000$ et $y = -111$, x et y comptant alors ensemble 6 bits non nuls.

Nous pouvons maintenant formuler l'algorithme de multiplication exacte TwoProd qui nécessite 17 opérations flottantes.

Algorithme 3.9. Transformation exacte du produit de deux nombres flottants.

function $[x, y] = \mathsf{TwoProd}(a, b)$
$x = a \otimes b$
$[a_h, a_l] = \mathsf{Split}(a)$
$[b_h, b_l] = \mathsf{Split}(b)$
$y = a_l \otimes b_l \ominus (((x \ominus a_h \otimes b_h) \ominus a_l \otimes b_h) \ominus a_h \otimes b_l))$

Remarque. Supposons que dans une boucle de nombreux appels à $\mathsf{TwoProd}(a, b)$ soient effectués, mais avec l'une des entrées constante, a par exemple. On a tout intérêt alors à calculer $[a_h, a_l] = \mathsf{Split}(a)$ une fois pour toute, avant que ne débute l'exécution de la boucle. Dans ce cas, nous noterons $\mathsf{TwoProd}_{a_h,a_l}(a, b)$ la transformation exacte de la multiplication à l'aide de l'algorithme 3.9. Néanmoins, un appel à $\mathsf{TwoProd}_{ah,al}(a, b)$ ne nécessite plus que 13 opérations flottantes.

En l'absence d'underflow, toutes les multiplications nécessaires au calcul de y à la dernière ligne de l'algorithme TwoProd sont exactes : dans ce cas, $\mathsf{TwoProd}(a, b)$ permet effectivement le calcul du produit exact $a \times b$. Mais en cas d'underflow cette propriété n'est plus assurée. La relation (3.6) du théorème suivant prend en compte l'erreur absolue introduite par chaque multiplication en cas d'underflow. Rappelons que $\mathbf{v}$ désigne l'unité d'underflow, soit la valeur du plus petit flottant dénormalisé positif.

Théorème 3.10 ([70])**.** *Soient $a, b \in \mathbf{F}$ et $x, y \in \mathbf{F}$ vérifiant $[x, y] = \mathsf{TwoProd}(a, b)$ (algorithme 3.9). On suppose que le mode d'arrondi courant est l'arrondi au plus proche. Alors, en absence d'underflow,*

$$a \times b = x + y, \quad x = a \otimes b, \quad |y| \leq \mathbf{u}|x|, \quad |y| \leq \mathbf{u}|a \times b|, \tag{3.5}$$

et, en présence d'un underflow,

$$a \times b = x + y + 5\eta, \quad x = a \otimes b, \quad |y| \leq \mathbf{u}|x| + 5\eta, \quad |y| \leq \mathbf{u}|a \times b| + 5\eta, \tag{3.6}$$

avec $|\eta| \leq \mathbf{v}$.

Remarque. Boldo présente dans [10] des hypothèses plus fines que celles utilisées dans le théorème 3.10 pour garantir le fonctionnement de l'algorithme 3.9. Indiquons qu'il est entre autres nécessaire, pour utiliser algorithme 3.9 dans des bases plus grandes que 2, de supposer que le nombre de chiffres de la mantisse est un nombre pair. Boldo démontre également qu'il est possible de remplacer le terme 5η dans la relation (3.6) par $7\eta/2$ en base quelconque, et par 3η lorsque l'on travaille en base 2.

La transformation exacte du produit peut être implantée beaucoup plus simplement sur une architecture disposant d'un Fused-Multiply-and-Add. C'est le cas par exemple sur l'Itanium d'Intel ou sur l'architecture Power d'IBM. Étant donnés trois flottants $a, b, c \in \mathbf{F}$, le résultat de l'opération $\mathsf{FMA}(a, b, c)$ est l'arrondi, dans le mode d'arrondi courant, du résultat exact $a \times b + c$. En l'absence d'underflow, l'erreur d'arrondi $y = a \times b - a \otimes b \in \mathbf{F}$ peut être calculée de la manière suivante [46, 68, 69]

$$\mathsf{FMA}(a, b, -a \otimes b) = \mathrm{fl}(a \times b - a \otimes b) = y.$$

On peut alors formuler l'algorithme TwoProdFMA, qui permet la transformation exacte d'un produit à l'aide du FMA, en seulement 2 opérations flottantes.

Algorithme 3.11. Transformation exacte du produit à l'aide du FMA.

function $[x, y] = \mathsf{TwoProdFMA}(a, b)$
  $x = a \otimes b$
  $y = \mathsf{FMA}(a, b, -x)$

Théorème 3.12 ([70]). *Soient $a, b \in \mathbf{F}$ et $x, y \in \mathbf{F}$ vérifiant $[x, y] = \mathsf{TwoProdFMA}(a, b)$ (algorithme 3.11). Alors, en l'absence d'underflow,*

$$a \times b = x + y, \quad x = a \otimes b, \quad |y| \leq \mathbf{u}|x|, \quad |y| \leq \mathbf{u}|a \times b|. \tag{3.7}$$

3.2.3 Transformation exacte de la division

Soient $a, b \in \mathbf{F}$, avec $b \neq 0$, et soit $q = a \oslash b$ le résultat calculé de la division de a par b. Évidemment, l'erreur d'arrondi $a/b - q$ n'est en général pas représentable dans $\mathbf{F}$. Par contre, en l'absence d'underflow le reste $r = a - b \times q$ de cette division l'est [9, 11]. Une transformation exacte pour la division peut donc être écrite de la manière suivante,

$$a = b \times q + r, \quad \text{avec} \quad q = a \oslash b \in \mathbf{F} \quad \text{et} \quad r \in \mathbf{F}.$$

Les sorties q et r peuvent être calculées grâce à l'algorithme DivRem [80].

Algorithme 3.13. Transformation exacte pour la division de deux nombres flottants.

function $[q, r] = \mathsf{DivRem}(a, b)$
  $q = a \oslash b$
  $[x, y] = \mathsf{TwoProd}(q, b)$
  $r = (a \ominus x) \ominus y$

Telle que formulée ci-dessus, la transformation exacte DivRem requiert 20 opérations flottantes. Dans les environnements disposant d'un FMA, l'appel à TwoProd peut être remplacé par un appel à TwoProdFMA, ce qui ramène à 5 le décompte des opérations flottantes.

Théorème 3.14. *Soient $a, b \in \mathbf{F}$ and $q, r \in \mathbf{F}$ tels que $[q, r] = \mathsf{DivRem}(a, b)$. Alors, en l'absence d'underflow,*

$$a = b \times q + r, \quad q = a \oslash b, \quad |r| \leq \mathbf{u}|b \times q|, \quad |r| \leq \mathbf{u}|a|.$$

3.2.4 Transformation exacte du Fused-Multiply-and-Add

Soient $a, b, c \in \mathbf{F}$ trois flottants. L'erreur d'arrondi $(a \times b + c) - \mathsf{FMA}(a, b, c)$ qui entache le résultat calculé $\mathsf{FMA}(a, b, c)$ n'est pas, dans le cas général, représentable sous la forme d'un nombre flottant. Cette erreur d'arrondi est par contre représentable sous la forme de la somme de deux nombres flottants, lorsque le mode d'arrondi est au plus proche [12]. Ceci est démontré dans [12] par Boldo et Muller, qui exhibent la transformation exacte suivante pour le FMA.

Algorithme 3.15. Transformation exacte pour le FMA.

function $[x, y, z] = \mathsf{ThreeFMA}(a, b, c)$
  $x = \mathsf{FMA}(a, b, c)$
  $[u_1, u_2] = \mathsf{TwoProdFMA}(a, b)$
  $[\alpha_1, z] = \mathsf{TwoSum}(c, u_2)$
  $[\beta_1, \beta_2] = \mathsf{TwoSum}(u_1, \alpha_1)$
  $y = (\beta_1 \ominus x) \oplus \beta_2$

L'algorithme ThreeFMA nécessite 17 opérations flottantes. Ses propriétés numériques sont résumées par le théorème 3.16 ci-dessous.

Théorème 3.16 ([12])**.** *Soient $a, b, c \in \mathbf{F}$ et $x, y, z \in \mathbf{F}$ vérifiant $[x, y, z] = \mathsf{ThreeFMA}(a, b, c)$ (algorithme 3.15). Alors, en absence d'underflow,*

$$a \times b + c = x + y + z, \quad x = \mathsf{FMA}(a, b, c), \quad |y + z| \leq \mathbf{u}|x|, \quad |y + z| \leq \mathbf{u}|a \times b + c| \tag{3.8}$$

et

$$y = 0 \quad ou \quad |y| > |z|. \tag{3.9}$$

3.3 Sommation et produit scalaire compensés

À titre d'exemples d'application des transformations exactes pour les opérations élémentaires, et parce qu'ils seront utilisés dans les chapitres 8 et 9, nous reproduisons dans cette section quelques résultats présents dans [70] à propos de la sommation et du produit scalaire compensés.

Précisons que les méthodes de sommations compensées sont depuis longtemps étudiées dans la littérature. Parmi les références antérieures à [70], citons les algorithmes de Kahan (1965) [45], Møller (1965) [65, 64], Pichat (1972) [78] et de Neumaier (1974) [67].

De nombreuses autres références existent pour des méthodes de sommation en arithmétique flottante [59, 60, 48, 37, 26, 23, 24, 62], et sur les algorithmes de sommation compensée en particulier [82, 2, 91, 90].

3.3.1 Sommation compensée

On considère un vecteur $p = (p_1, \ldots, p_n) \in \mathbf{R}^n$, et on note $s = \sum_{i=1}^n p_i$ et $\widetilde{s} = \sum_{i=1}^n |p_i|$. Le nombre de conditionnement pour le problème du calcul de la somme des n valeurs $p_1, \ldots, p_n$ est défini dans [70] de la manière suivante,

$$\mathrm{cond}\left(\sum p\right) := \frac{\sum_{i=1}^n |p_i|}{\sum_{i=1}^n p_i} = \frac{\widetilde{s}}{|s|}. \tag{3.10}$$

Soit maintenant $p = (p_1, \ldots, p_n) \in \mathbf{F}^n$ un vecteur de n flottants. On considère ci-dessous l'algorithme classique de sommation récursive.

Algorithme 3.17. Sommation récursive de n flottants

function $\widehat{s}_n = \mathsf{Sum}(p)$
  $\widehat{s}_1 = p_1$
  for $i = 2 : n$
    $\widehat{s}_i = \widehat{s}_{i-1} \oplus p_i$
  end

Pour cet algorithme de sommation, notons σ_{i-1} l'erreur d'arrondi commise lors de l'addition flottante $\widehat{s}_{i-1} \oplus p_i$. D'après le théorème 3.6, on sait que $\sigma_{i-1} \in \mathbf{F}$ et $\widehat{s}_{i-1} + p_i = \widehat{s}_i + \sigma_{i-1}$. Ainsi on a

$$p_i = \widehat{s}_i - \widehat{s}_{i-1} + \sigma_{i-1}.$$

Comme $\widehat{s}_1 = p_1$, on peut montrer par récurrence que

$$\sum_{i=1}^{n} p_i = \widehat{s}_n + \sum_{i=0}^{n-1} \sigma_i.$$

En notant de plus $\sigma_n = \widehat{s}_n = \mathsf{Sum}(p)$, on obtient l'égalité suivante,

$$\sum_{i=1}^{n} p_i = \sum_{i=1}^{n} \sigma_i, \quad \text{avec} \quad \sigma_n = \mathsf{Sum}(p), \tag{3.11}$$

Le vecteur $\sigma = (\sigma_1, \ldots, \sigma_n) \in \mathbf{F}^n$ peut être calculé, comme indiqué dans [70], par l'algorithme VecSum ci-dessous.

Algorithme 3.18. Transformation exacte pour la somme de n flottants
(Algorithme 4.4 dans [70]).

function $\sigma = \mathsf{VecSum}(p)$
  $\widehat{s}_1 = p_1$
  for $i = 2 : n$
    $[\widehat{s}_i, \sigma_{i-1}] = \mathsf{TwoSum}(\widehat{s}_{i-1}, p_i)$
  end
  $\sigma_n = \widehat{s}_n$

La relation (3.11) démontre que l'algorithme VecSum est une transformation exacte la sommation de n nombre flottants. Cette transformation exacte est utilisée de la manière suivante pour compenser l'algorithme de sommation récursive [70].

Algorithme 3.19. Somme compensée de n flottants
(Algorithme 4.4 dans [70]).

function $\overline{s} = \mathsf{Sum2}(p)$
  $\sigma = \mathsf{VecSum}(p)$
  $\widehat{c} = \mathsf{Sum}\big(\sigma_1, \ldots, \sigma_{n-1}\big)$
  $\overline{s} = \sigma_n \oplus \widehat{c}$

Puisque $\sigma_n = \mathsf{Sum}(p)$, et d'après l'égalité (3.11), l'idée de l'algorithme Sum2 est de calculer une valeur approchée $\widehat{c}$ de $\sum_{i=1}^{n-1} \sigma_i$, puis d'en déduire un résultat compensé $\overline{s} = \sigma_n \oplus \widehat{c}$. Puisque le terme correctif $\widehat{c}$ prend en compte les erreurs d'arrondi générées lors du calcul de $\sigma_n = \mathsf{Sum}(p)$, on peut s'attendre à ce que le résultat compensé $\overline{s}$ soit plus précis que le résultat initial $\mathsf{Sum}(p)$. Les auteurs de [70] démontrent en effet le théorème suivant.

Théorème 3.20 (proposition 4.4 dans [70]). *Étant donné un vecteur $p = (p_1, \ldots, p_n)$ de n nombres flottants, soit $s = \sum_{i=1}^{n} p_i$ et soit $\widetilde{s} = \sum_{i=1}^{n} |p_i|$. On suppose que $n\mathbf{u} < 1$, et on note $\overline{s}$ le flottant tel que $\overline{s} = \mathsf{Sum2}(p)$. Alors, même en présence d'underflow,*

$$|s - \overline{s}| \leq \mathbf{u}|s| + \gamma_{n-1}^2 \widetilde{s}. \tag{3.12}$$

Pour interpréter la borne d'erreur (3.12), on fait intervenir le nombre de conditionnement (3.10),

$$\frac{|\overline{s} - s|}{|s|} \leq \mathbf{u} + \gamma_{n-1}^2 \operatorname{cond}\left(\sum p\right). \tag{3.13}$$

Le premier terme $\mathbf{u}$, dans borne (3.13) sur l'erreur relative entachant le résultat calculé $\overline{s}$ reflète l'arrondi final vers la précision de travail. Le second terme est essentiellement de l'ordre de $n^2\mathbf{u}^2$ fois le conditionnement de la somme. Ce résultat indique donc que la qualité du résultat compensé $\overline{s}$ est la même que s'il avait été calculé par la méthode de sommation récursive classique en précision doublée $\mathbf{u}^2$, avec un arrondi final vers la précision $\mathbf{u}$.

Les auteurs de [70] généralisent ensuite cette méthode en appliquant $K - 1$ fois la transformation exacte VecSum. Cela donne l'algorithme SumK, qui effectue $(6K-5)(n-1)$ opérations flottantes.

Algorithme 3.21. Sommation compensée $K - 1$ fois.

function $\overline{s} = \mathsf{SumK}(p, K)$
  for $k = 1 : K - 1$
    $p = \mathsf{VecSum}(p)$
  end
  $c = \mathsf{Sum}\big(p_1, \ldots, p_{n-1}\big)$
  $\overline{s} = p_n \oplus c$

Le théorème suivant montre que le résultat calculé par l'algorithme SumK est maintenant aussi précis que si la somme avait été calculée en K fois la précision de travail.

Théorème 3.22 (proposition 4.10 dans [70]). *Étant donné un vecteur $p = (p_1, \ldots, p_n)$ de n nombres flottants, soit $s = \sum_{i=1}^{n} p_i$ et soit $\widetilde{s} = \sum_{i=1}^{n} |p_i|$. On suppose que $4n\mathbf{u} \leq 1$ et $K \geq 3$. On note $\overline{s}$ le flottant tel que $\overline{s} = \mathsf{SumK}(p, K)$. Alors, même en présence d'underflow,*

$$|\overline{s} - s| \leq (\mathbf{u} + 3\gamma_{n-1}^2)|s| + \gamma_{2n-2}^K \widetilde{s}. \tag{3.14}$$

À nouveau, ce théorème s'interprète à l'aide du nombre de conditionnement (3.10). La précision du résultat compensé $\overline{s}$ calculé par SumK est majorée de la façon suivante,

$$\frac{|\overline{s} - s|}{|s|} \leq (\mathbf{u} + 3\gamma_{n-1}^2) + \gamma_{2n-2}^K \operatorname{cond}\left(\sum p\right). \tag{3.15}$$

Le facteur γ_{2n-2}^K, de l'ordre de $4n^K\mathbf{u}^K$, indique maintenant que les calculs sont aussi précis que s'ils avaient été calculée en K fois la précision de travail [70]. Le terme $\mathbf{u} + 3\gamma_{n-1}^2$, de l'ordre de $\mathbf{u}$ reflète simplement l'arrondi final vers la précision de travail.

3.3.2 Produit scalaire compensé

Nous passons plus rapidement sur le cas du produit scalaire compensé, qui est très similaire à celui de la somme compensée. L'algorithme Dot2 rappelé ici sera utilisé au chapitre 9.

Soient $x = (x_1, \ldots, x_n)^T \in \mathbf{R}^n$ et $x = (y_1, \ldots, y_n)^T \in \mathbf{R}^n$ deux vecteurs de longueur n. On note $s = x^T y$ le produit scalaire de x et de y, et $\widetilde{s} = |x^T||y| = \sum_{i=1}^{n} |x_i y_i|$. Le nombre de conditionnement pour le problème du calcul du produit scalaire $x^T y$ est défini de la manière suivante dans [70],

$$\mathrm{cond}(x^T y) := 2\frac{|x^T||y|}{|x^T y|} = \frac{\widetilde{s}}{|s|}. \tag{3.16}$$

On suppose maintenant que $x = (x_1, \ldots, x_n)^T \in \mathbf{F}^n$ et $x = (y_1, \ldots, y_n)^T \in \mathbf{F}^n$ sont deux vecteurs de flottants. Les auteurs de [70] considèrent l'algorithme de produit scalaire suivant, dans lequel les erreurs commises lors du calcul classique du produit scalaire sont compensées à l'aide de TwoProd et de TwoSum.

Algorithme 3.23. Produit scalaire compensé de deux vecteurs de n flottants.

function $\overline{s} = \mathsf{Dot2}(x, y)$
  $[\widehat{s}_1, c_1] = \mathsf{TwoProd}(x_1, y_1)$
  for $i = 2 : n$
    $[\widehat{p}_i, \sigma_i] = \mathsf{TwoProd}(x_i, y_i)$
    $[\widehat{s}_i, \sigma_i] = \mathsf{TwoSum}(\widehat{s}_{i-1}, \widehat{p}_i)$
    $\widehat{c}_i = \widehat{c}_{i-1} \oplus (\pi_i \oplus \sigma_i)$
  end
  $\overline{s} = \widehat{s}_n \oplus \widehat{c}_n$

Théorème 3.24 (proposition 5.5 dans [70])**.** *Étant donnés $x = (x_1, \ldots, x_n)^T \in \mathbf{F}^n$ et $x = (y_1, \ldots, y_n)^T \in \mathbf{F}^n$ sont deux vecteurs de flottants, soit $\overline{s} = \mathsf{Dot2}(x, y)$ la valeur approchée de $x^T y$ calculée par algorithme 3.23. On suppose que $n\mathbf{u} \leq 1$. Alors, en l'absence d'underflow,*

$$|\overline{s} - s| \leq \mathbf{u}|x^T y| + \gamma_n^2 |x^T||y|. \tag{3.17}$$

En faisant intervenir le nombre de conditionnement défini par la relation (3.16), on constate facilement que la borne d'erreur (3.17) signifie à nouveau que le résultat compensé calculé par Dot2 est aussi précis que s'il avait été calculé par l'algorithme classique de produit scalaire en précision doublée $\mathbf{u}^2$, puis arrondi vers la précision de travail $\mathbf{u}$.

3.4 Arithmétiques doubles-doubles et quad-double

Telle que définie par Priest [81, 82], une expansion de longueur n est la représentation d'un réel $\mathbf{x}$ sous la forme d'un n-uplet $(x_1, \ldots, x_n) \in \mathbf{F}^n$ tel que

1. $\mathbf{x}$ est égal à la somme exacte, non évaluée, des x_i ;
2. les x_i non nuls sont triés par ordre décroissant de valeur absolue ;
3. les x_i non nuls ne se chevauchent pas : x_i et x_{i+1} ne présentent aucun bit significatif de même poids.

Les x_i sont communément appelées les composantes de l'expansion $\mathbf{x}$. Priest propose également des algorithmes pour effectuer exactement les opérations arithmétiques élémentaires sur les expansions — addition et produit de deux expansions, division d'une expansion par une autre. Ces algorithmes permettent donc d'effectuer des calculs en précision arbitraire à l'aide de l'arithmétique flottante disponible sur les machines, de précision nécessairement finie. Cet auteur décrit également une procédure de renormalisation, permettant d'assurer que la propriété de non chevauchement est bien vérifiée par l'expansion produite par une opération arithmétique élémentaire.

Notons que les algorithmes formulés par Priest sont valables dans le cadre d'une arithmétique flottante avec arrondi fidèle. Shewchuk [87, 88] adaptera ensuite les travaux de Priest dans la cadre de la norme IEEE-754, en supposant en particulier que les opérations de d'arithmétiques flottantes sont toutes effectuées en arrondi au plus proche.

Les expansions de longueur fixe constituent un cas particulier de travaux relatés ci-dessus : dans ce cas, on fixe à l'avance le nombre de composantes utilisées par expansion, ce qui permet de simuler une précision de travail fixée. Citons ici les auteurs de [35] :

> We note that many applications would get full benefit from using merely a small multiple of (such as twice or quadruple) the working precision, without the need for arbitrary precision. The algorithms for this kind of "fixed" precision can be made significantly faster than those for arbitrary precision. Bailey [4] and Briggs [14] have developed algorithms and software for "double-double" precision, twice the double precision[1]. They used the multiple-component format, where a double-double number is represented as an unevaluated sum of a leading double and a trailing double.

Il est également à noter que de nombreux travaux antérieurs à ceux évoqués ci-dessus visent à simuler une précision de travail doublée à l'aide d'un format de représentation des nombres proche de celui des expansions de longueur deux. C'est la cas par exemple des travaux de Møller [65, 64], Dekker [19], Knuth [47] et Linnainmaa [58].

Dans tous les cas cités ci-dessus, les briques de bases permettant les opérations élémentaires sur les expansions sont ce que nous appelons dans ce document des transformations arithmétiques exactes, comme celles décrites dans la section précédente.

Dans la suite de cette section, nous fournissons quelques détails sur l'arithmétique « double-double » [4, 34, 57] : nous décrivons notamment les algorithmes qui nous seront utiles pour comparer les performances de l'arithmétique double-double à celles des algorithmes compensés que nous présenterons par la suite — nous effectuerons notamment de telles comparaisons au chapitre 5. Nous dirons également quelques mots à propos de l'arithmétique quad-double [35].

3.4.1 Arithmétique double-double

Les opérations arithmétiques sur les doubles-doubles permettent de doubler la double précision IEEE-754, offrant ainsi précision de travail de l'ordre de 106 bits. Ce doublement de la précision est obtenu uniquement à l'aide d'opérations arithmétiques élémentaires, dans le mode d'arrondi au plus proche. Le format double-double tel que nous le présentons ici est celui décrit dans [57] et utilisé dans [34]. Un double-double $\mathbf{a}$ est une paire (ah, al), formée de deux flottants double précision IEEE-754, telle que :

[1] double précision IEEE-754 [41].

1. $a = ah + al$, et
2. $ah \oplus al = ah$.

Le fait que l'on impose $ah \oplus al = ah$ assure que les flottants ah et al sont triés par ordre de valeurs absolues décroissantes. Puisque l'on travaille en arrondi au plus proche, en double précision IEEE-754, la condition $ah \oplus al = ah$ implique $|al| \leq 1/2\,\mathrm{ulp}(ah)$. De plus, lorsque $|al| = 1/2\,\mathrm{ulp}(ah)$, la règle de l'arrondi pair implique que le bit de poids faible de ah est nul. La condition $ah \oplus al = ah$ assure donc aussi que les flottants ah et al ne se chevauchent pas. Comme nous l'avons déjà dit, cela implique une *étape de renormalisation* à la fin de chaque opération arithmétique entre double-doubles.

Dans nos comparaisons entre l'arithmétique double-double et les algorithmes compensés, nous n'aurons besoin que de deux opérations :

1. l'addition d'un double-double et d'un double,
2. le produit d'un double-double par un double.

Nous formulons ci-dessous les algorithmes permettant d'effectuer ces opérations. Comme dans [57], nous supposons que les opérations flottantes sont effectuées en double précision IEEE-754, dans le mode d'arrondi au plus proche.

L'algorithme add_dd_d ci-dessous calcule l'addition d'un double-double et d'un double, en 10 opérations flottantes.

Algorithme 3.25. Addition $(rh, rl) = (ah, al) \oplus b$

function $[rh, rl] = \mathsf{add\_dd\_d}(ah, al, b)$
$[th, \delta] = \mathsf{TwoSum}(a_h, b)$
$tl = al \oplus \delta$
$[rh, rl] = \mathsf{FastTwoSum}(th, tl)$

L'algorithme prod_dd_d effectue le produit d'un double-double par un double, en 22 opérations flottantes.

Algorithme 3.26. Produit $(rh, rl) = (ah, al) \otimes b$.

function $[rh, rl] = \mathsf{prod\_dd\_d}(ah, al, b)$
$[th, \delta] = \mathsf{TwoProd}(ah, b)$
$tl = al \otimes b \oplus \delta$
$[rh, rl] = \mathsf{FastTwoSum}(th, tl)$

Si un FMA est disponible, on utilise cette instruction pour le calcul de tl, et on remplace bien sûr TwoProd par TwoProdFMA : le décompte des opérations flottantes se trouve alors ramené à 7.

Dans chacun des algorithmes formulés ci-dessus, le fait que la paire (rh, rl) produite en sortie soit toujours une expansion de longueur deux, c'est à dire que $|rl| \leq \frac{1}{2}\,\mathrm{ulp}(rh)$, est assuré par la dernière instruction $[rh, rl] = \mathsf{FastTwoSum}(th, tl)$. Nous désignerons par la suite cette instruction sous le nom d'*étape de renormalisation*.

À titre d'exemple, et parce que nous y ferons souvent référence au cours des chapitres suivants, nous formulons ci-dessous le schéma de Horner (algorithme 2.6) en effectuant les calculs internes à l'aide de l'arithmétique double-double. On considère donc un polynôme

$p(x) = \sum_{i=0}^{n} a_i x^i$ à coefficients flottants, et x un flottant. Les opérations de l'arithmétique double-double sont ici *inlinées*, afin de détailler la structure de l'algorithme : ces deux opérations sont encadrées dans l'algorithme 3.27. Nous utilisons un appel à $\mathsf{TwoProd}_{xh,xl}(sh_{i+1}, x)$ au lieu de $\mathsf{TwoProd}(sh_{i+1}, x)$, afin que le découpage de x ne soit effectué qu'une seule fois avant le début de la boucle.

L'algorithme DDHorner nécessite $29n + (1)$ opérations flottantes. Lorsqu'un FMA est disponible sur la machine cible, TwoProd sera bien entendu remplacé par TwoProdFMA, et la ligne $tl = sl_{i+1} \otimes x \oplus tl$ par $tl = \mathsf{FMA}(sl_{i+1}, x, tl)$. En présence d'un FMA, l'algorithme DDHorner ne nécessite donc plus que $16n + \mathcal{O}(1)$ opérations flottantes.

Algorithme 3.27. Algorithme de Horner en arithmétique double-double

function $r = \mathsf{DDHorner}(p, x)$
$[xh, xl] = \mathsf{Split}(x)$
$sh_n = a_i$; $sl_n = 0$
for $i = n - 1 : -1 : 0$

$\%$ double-double = double-double $\times$ double :
$\%$ $(ph_i, pl_i) = (sh_{i+1}, sl_{i+1}) \otimes x$
$[th, tl] = \mathsf{TwoProd}_{xh,xl}(sh_{i+1}, x)$
$tl = sl_{i+1} \otimes x \oplus tl$
$[ph_i, pl_i] = \mathsf{FastTwoSum}(th, tl)$

$\%$ double-double = double-double + double :
$\%$ $(sh_i, sl_i) = (ph_i, pl_i) \oplus a_i$
$[th, tl] = \mathsf{TwoSum}(ph_i, a_i)$
$tl = tl \oplus pl_i$
$[sh_i, sl_i] = \mathsf{FastTwoSum}(th, tl)$

end
$r = sh_0$

3.4.2 Arithmétique quad-double

Par extension de la définition d'un double-double, un quad-double **a** est un quadruplet (a_0, a_1, a_2, a_3) de flottants double precision IEEE-754 tel, que

1. $a = a_0 + a_1 + a_2 + a_3$, et
2. $a_i \oplus a_{i+1} = a_i$, pour $i = 0, 1, 2$.

Nous ne présentons pas ici les algorithmes introduits dans [35] : les idées sont les mêmes que dans le cas de l'arithmétique double-double, mais les algorithmes sont plus complexes. Précisons néanmoins que les opérations arithmétique définies dans cet article permettent de simuler quatre fois la double précision IEEE-754, ce qui donne une précision de travail de l'ordre de 212 bits. Comme dans le cas des doubles-doubles, les algorithmes sur les quad-doubles n'effectuent que des opérations élémentaires de l'arithmétique flottante IEEE-754, dans le mode d'arrondi au plus proche.

3.5 Conclusions

Dans ce chapitre, nous avons passé en revue les principales transformation exactes connues pour les opérations élémentaires de l'arithmétique flottante. Nous avons également cité deux exemples de leur utilisation dans la littérature.

Au travers des résultats de Ogita, Rump et Oishi, nous avons illustré l'utilisation de ces transformations exactes pour compenser les erreurs d'arrondi générées par les algorithmes de sommation récursive et de produit scalaire. Nous développerons cette technique au cours des chapitres suivants, dans le cas de l'évaluation de polynômes et dans celui de la résolution de systèmes linéaires triangulaires.

Nous avons également rappelé l'utilisation des transformations exactes au sein des bibliothèques double-double et quad-double, simulant une précision de travail équivalente respectivement à deux fois et quatre fois la double précision IEEE-754. Ces bibliothèques seront fréquemment utilisées dans nos tests de performances, afin de mettre en évidence l'efficacité pratique des algorithmes compensés.

Dans la suite de ce document, nous supposerons toujours que l'on utilise l'arithmétique flottante IEEE-754, dans le mode d'arrondi au plus proche. Cette hypothèse assure que les erreurs d'arrondi élémentaires peuvent être représentées dans $\mathbf{F}$, ainsi que la validité des transformations exactes décrites dans ce chapitre.

Il existe dans la littérature des résultats basés sur des hypothèses plus fines pour garantir le fait que les erreurs d'arrondi puissent être représentées et calculées [11, 10]. Notons également que Priest [82] définit des transformations exactes pour la multiplication dans le cadre plus large d'une arithmétique flottante avec arrondi fidèle. Il serait intéressant d'étudier ce type de résultat afin de proposer l'utilisation des algorithmes compensés sur des architectures ne disposant pas de l'arrondi au plus proche. Citons par exemple le cas du processeur Cell d'IBM, qui dispose d'une très grande puissance de calcul en simple précision IEEE-754, mais uniquement dans le mode d'arrondi vers zéro.

Chapitre 4

Schéma de Horner compensé

Plan du chapitre : Nous introduirons d'abord une transformation exacte pour l'évaluation par l'algorithme de Horner en Section 4.2 : cette transformation exacte nous permettra de proposer l'algorithme d'évaluation compensé CompHorner. En Section 4.3, nous prouvons une borne sur l'erreur entachant ce résultat compensé. Cette borne montre que l'algorithme CompHorner est aussi précis que le schéma de Horner classique utilisé en précision doublée, avec un arrondi final vers la précision courante. En particulier, l'erreur relative entachant le résultat compensé est de l'ordre de l'unité d'arrondi tant que le conditionnement de l'évaluation est petit devant $\mathbf{u}^{-1}$. En Section 4.4, nous justifions clairement ce fait, en proposant une borne supérieure sur le conditionnement permettant d'assurer que le résultat compensé est un arrondi fidèle du résultat exact. Les expériences de la Section 4.5 illustrent le comportement numérique de CompHorner. Pour cela nous détaillerons une méthode pour générer des polynômes à coefficients flottants de conditionnement arbitrairement grand. Les résultats des Sections 4.2 à 4.4 sont démontrés en supposant qu'aucun résultat dénormalisé n'intervient au cours des calculs. En Section 4.6 nous proposons donc une nouvelle borne d'erreur pour le résultat compensé calculé par CompHorner, valable également en présence d'underflow.

4.1 Conditionnement et précision de l'évaluation polynomiale

Les polynômes apparaissent dans de nombreux domaines du calcul scientifique et de l'ingénierie. D'autre part, développer des algorithmes numériques rapides et fiables pour les évaluer reste un grand challenge. Les méthodes de calcul numérique de racines se basent en général sur des méthodes itératives de type Newton, qui nécessitent d'évaluer un polynôme et ses dérivées. Higham [39, chap. 5] consacre, par exemple, un chapitre entier aux polynômes et en particulier à l'évaluation polynomiale.

Si $p(x) = \sum_{i=0}^{n} a_i x^i$ est un polynôme dont les coefficients sont des nombres flottants, et x est un argument flottant, alors la précision relative du résultat $\widehat{p}(x)$ calculé par le schéma de Horner vérifie l'estimation

$$\frac{|p(x) - \widehat{p}(x)|}{|p(x)|} \leq \mathrm{cond}(p, x) \times \mathcal{O}(\mathbf{u}). \tag{4.1}$$

Dans l'inégalité précédente, $\mathrm{cond}(p,x)$ est le nombre de conditionnement de l'évaluation de $p(x)$, qui quantifie la difficulté d'une évaluation précise du résultat. En notant

$$\widetilde{p}(x) := \sum_{i=0}^{n} |a_i||x|^i, \tag{4.2}$$

le conditionnement de l'évaluation de p en x est classiquement défini par [20]

$$\mathrm{cond}(p,x) = \frac{\widetilde{p}(x)}{|p(x)|}. \tag{4.3}$$

On notera que le nombre de conditionnement $\mathrm{cond}(p,x)$ peut être arbitrairement grand, par exemple au voisinage des racines de p.

Ici nous ne considérerons que des polynômes dont les coefficients sont des nombres flottants. Cette situation apparaît par exemple lors de l'évaluation de fonctions élémentaires [66], ou en calcul géométrique [55]. Le problème est que même dans ce cas particulier où les coefficients de p sont des flottants, la relation (4.1) montre que le résultat calculé par le schéma de Horner peut être extrêmement moins précis que l'unité d'arrondi $\mathbf{u}$. C'est le cas lorsque $\mathrm{cond}(p,x)$ est grand devant $\mathbf{u}^{-1}$, et on parlera alors d'évaluation mal conditionnée.

Lorsque la précision de travail $\mathbf{u}$ n'est pas suffisante pour garantir la précision du résultat, plusieurs alternatives existent pour simuler des précisions de travail supérieures. Au chapitre précédent nous avons rappelé que les expansions de longueur fixe, telles les « double-double » et les « quad-double » [81, 57] sont des solutions logicielles couramment utilisées pour simuler respectivement une précision double ou quadruple de la double précision IEEE-754.

Nous proposons dans ce chapitre une alternative à l'utilisation de cette arithmétique double-double, dans le contexte de l'évaluation de polynômes à une indéterminée. Nous supposons que tous les calculs sont effectués à une précision fixée $\mathbf{u}$, dans le mode d'arrondi au plus proche. Le principe du schéma de Horner compensé que nous présentons ici est celui de la compensation des erreurs d'arrondi, à l'aide des transformations exactes décrites au chapitre 3. Nous montrerons que la précision de l'évaluation calculée par le schéma de Horner compensé satisfait

$$\frac{|p(x) - \widehat{p}(x)|}{|p(x)|} \leq \mathbf{u} + \mathrm{cond}(p,x) \times \mathcal{O}(\mathbf{u}^2). \tag{4.4}$$

Comparée à la relation (4.1), l'inégalité précédente montre que le résultat de l'évaluation compensé est aussi précis que celui calculé par le schéma de Horner classique en précision doublée, avec un arrondi final vers la précision de travail. Nous avons déjà mentionné ce comportement au chapitre précédent au sujet de la sommation et du produit scalaire compensés.

Comme dans le cas du schéma de Horner classique, la borne d'erreur relative (4.4) dépend du nombre de conditionnement de l'évaluation : la précision du résultat compensé peut donc être arbitrairement mauvaise pour des évaluations mal conditionnées. Néanmoins, la borne (4.4) montre que la précision du résultat compensé est du même ordre de grandeur que la précision des calculs tant que $\mathrm{cond}(p,x)$ est petit devant $\mathbf{u}^{-1}$.

Cette remarque motive une partie de notre étude du schéma de Horner compensé, portant sur le calcul d'un arrondi fidèle de $p(x)$. Rappelons que $\overline{r}$ est un arrondi fidèle

de l'évaluation exacte $p(x)$, lorsque soir $\overline{r} = p(x)$, soit $\overline{r}$ est l'un des deux flottants les plus proches de $p(x)$. Nous proposons dans ce chapitre une condition suffisante permettant d'assurer le fait que le résultat compensé soit un arrondi fidèle de $p(x)$. Ce résultat peut être interprété de la manière suivante : tant que le conditionnement est petit devant $\mathbf{u}^{-1}$, dans un sens précisé clairement par le théorème 4.10, le schéma de Horner compensé calcule un arrondi fidèle du résultat exact.

Les résultats évoqués jusqu'à présent sont valides en supposant qu'aucun underflow n'intervient au cours des calculs. Nous fournirons finalement dans ce chapitre une borne d'erreur *a priori* pour le résultat calculé par le schéma de Horner compensé, vérifiée également en présence d'underflow.

4.2 Du schéma de Horner à sa version compensée

Comme nous le verrons plus en détail par la suite, le principe du schéma de Horner compensé est de calculer un terme correctif afin de compenser l'effet des erreurs d'arrondis générées par le schéma de Horner classique. Le but de cette section est de montrer comment un tel terme correctif peut être calculé, et de formuler l'algorithme de Horner compensé.

4.2.1 Une transformation exacte pour le schéma de Horner

L'algorithme suivant calcule, en même temps que l'évaluation de $p(x)$ par le schéma de Horner, deux polynômes qui permettent d'exprimer exactement l'erreur directe entachant cette évaluation. Comme le montre le théorème 4.2, cet algorithme sera donc vu comme une transformation exacte pour le schéma de Horner.

Algorithme 4.1. Transformation exacte pour le schéma de Horner

function $[r_0, p_\pi, p_\sigma] = \mathsf{EFTHorner}(p, x)$
  $[xh, xl] = \mathsf{Split}(x)$
  $r_n = a_n$
  for $i = n-1 : -1 : 0$
    $[p_i, \pi_i] = \mathsf{TwoProd}_{xh,xl}(r_{i+1}, x)$
    $[r_i, \sigma_i] = \mathsf{TwoSum}(p_i, a_i)$
    Soit π_i le coefficient de degré i du polynôme p_π
    Soit σ_i le coefficient de degré i du polynôme p_σ
  end

L'algorithme EFTHorner nécessite $19n + 4$ opérations flottantes. Le théorème suivant montre que l'algorithme 4.1 est bien une transformation exacte pour l'évaluation polynomiale avec le schéma de Horner.

Théorème 4.2. *Soit $p(x) = \sum_{i=0}^{n} a_i x^i$ un polynôme de degré n à coefficients flottants, et soit x un nombre flottant. En l'absence d'underflow, l'algorithme 4.1 calcule à la fois $\mathsf{Horner}(p, x)$ et deux polynômes p_π et p_σ de degré $n-1$ à coefficients flottants, tels que*

$$p(x) = \mathsf{Horner}(p, x) + (p_\pi + p_\sigma)(x). \tag{4.5}$$

De plus,

$$\widetilde{(p_\pi + p_\sigma)}(x) \leq (\widetilde{p}_\pi + \widetilde{p}_\sigma)(x) \leq \gamma_{2n}\widetilde{p}(x). \tag{4.6}$$

Preuve. Comme TwoProd et TwoSum sont des transformations exactes, d'après les théorèmes 3.10 et 3.6 on a $r_{i+1}x = p_i + \pi_i$ et $p_i + a_i = r_i + \sigma_i$. Ainsi, pour $i = 0, \ldots, n-1$ on a l'égalité $r_i = r_{i+1}x + a_i - \pi_i - \sigma_i$. Comme de plus $r_n = a_n$, on peut montrer par récurrence qu'à la fin de la boucle

$$r_0 = \sum_{i=0}^{n} a_i x^i - \sum_{i=0}^{n-1} (\pi_i + \sigma_i) x^i,$$

ce qui démontre la relation (4.5).

Afin de prouver la relation (4.6), démontrons par récurrence sur $i = 1, \ldots, n$ que

$$|p_{n-i}| \leq (1+\gamma_{2i-1}) \sum_{j=1}^{i} |a_{n-i+j}||x^j|, \quad \text{et} \tag{4.7}$$

$$|r_{n-i}| \leq (1+\gamma_{2i}) \sum_{j=0}^{i} |a_{n-i+j}||x^j|. \tag{4.8}$$

Pour $i = 1$, comme $r_n = a_n$, on a

$$|p_{n-1}| = |a_n \otimes x| \leq (1+\mathbf{u})|a_n||x| \leq (1+\gamma_1)|a_n||x|,$$

donc (4.7) est vraie. De la même façon, comme

$$|r_{n-1}| \leq (1+\mathbf{u}) \left((1+\gamma_1)|a_n||x| + |a_{n-1}|\right) \leq (1+\gamma_2) \left(|a_n||x| + |a_{n-1}|\right),$$

la relation (4.8) est donc aussi vérifiée. Supposons maintenant que (4.7) et (4.8) sont vraies pour un entier i tel que $1 \leq i < n$. Puisque $|p_{n-(i+1)}| = |r_{n-i} \otimes x| \leq (1+\mathbf{u})|r_{n-i}||x|$, de l'hypothèse de récurrence on déduit

$$|p_{n-(i+1)}| \leq (1+\mathbf{u})(1+\gamma_{2i}) \sum_{j=0}^{i} |a_{n-i+j}||x^{j+1}| \leq \left(1+\gamma_{2(i+1)-1}\right) \sum_{j=1}^{i+1} |a_{n-(i+1)+j}||x^j|.$$

D'autre part, on a $|r_{n-(i+1)}| = |p_{n-(i+1)} \oplus a_{n-(i+1)}| \leq (1+\mathbf{u})(|p_{n-(i+1)}| + |a_{n-(i+1)}|)$, d'où

$$|r_{n-(i+1)}| \leq (1+\mathbf{u}) \left(1+\gamma_{2(i+1)-1}\right) \sum_{j=0}^{i+1} |a_{n-(i+1)+j}||x^j| \leq \left(1+\gamma_{2(i+1)}\right) \sum_{j=0}^{i+1} |a_{n-(i+1)+j}||x^j|.$$

Les relations (4.7) et (4.8) sont donc démontrées par récurrence. Pour $i = 1, \ldots, n$, on écrit donc

$$|p_{n-i}||x^{n-i}| \leq (1+\gamma_{2i-1})\widetilde{p}(x), \quad \text{et} \quad |r_{n-i}||x^{n-i}| \leq (1+\gamma_{2i})\widetilde{p}(x).$$

Puisque $[p_i, \pi_i] = \mathsf{TwoProd}_{xh,xl}(r_{i+1}, x)$ et $[r_i, \sigma_i] = \mathsf{TwoSum}(p_i, a_i)$, d'après les théorèmes 3.10 et 3.6, on a $|\pi_i| \leq \mathbf{u}|p_i|$ et $|\sigma_i| \leq \mathbf{u}|r_i|$ pour $i = 0, \ldots, n-1$. Par conséquent,

$$(\widetilde{p}_\pi + \widetilde{p}_\sigma)(x) = \sum_{i=0}^{n-1} (|\pi_i| + |\sigma_i|)|x^i| \leq \mathbf{u} \sum_{i=1}^{n} (|p_{n-i}| + |r_{n-i}|)|x^{n-i}|,$$

et donc

$$(\widetilde{p}_\pi + \widetilde{p}_\sigma)(x) \leq \mathbf{u} \sum_{i=1}^{n} (2 + \gamma_{2i-1} + \gamma_{2i})\, \widetilde{p}(x) \leq 2n\mathbf{u}\,(1+\gamma_{2n})\, \widetilde{p}(x).$$

Puisque $2n\mathbf{u}(1+\gamma_{2n}) = \gamma_{2n}$, on obtient finalement $(\widetilde{p}_\pi + \widetilde{p}_\sigma)(x) \leq \gamma_{2n}\widetilde{p}(x)$. ■

4.2.2 Schéma de Horner compensé

Étant donné un polynôme p à coefficients flottants, et x un flottant, on considère l'évaluation $\widehat{r}$ de $p(x)$ calculée à l'aide du schéma de Horner. D'après le théorème 4.2, l'erreur directe entachant $\widehat{r}$ est exactement

$$c = p(x) - \widehat{r} = (p_\pi + p_\sigma)(x),$$

où les deux polynômes p_π et p_σ sont identifiés exactement à l'aide de l'algorithme 4.1. Le principe du schéma de Horner compensé est de calculer une approximation $\widehat{c}$ de cette erreur directe c, puis d'en déduire un résultat compensé $\overline{r}$, tel que

$$\overline{r} = \widehat{r} \oplus \widehat{c}.$$

Par la suite, nous dirons que c est le terme correctif exact pour $\widehat{r}$, et que $\widehat{c}$ est terme correctif calculé. Nous pouvons maintenant formuler l'algorithme de Horner compensé.

Algorithme 4.3. Schéma de Horner compensé

function $\overline{r} = \mathsf{CompHorner}(p, x)$
  $[\widehat{r}, p_\pi, p_\sigma] = \mathsf{EFTHorner}(p, x)$
  $\widehat{c} = \mathsf{Horner}(p_\pi \oplus p_\sigma, x)$
  $\overline{r} = \widehat{r} \oplus \widehat{c}$

Il est important de noter que si l'on avait $\widehat{c} = c$, alors on aurait également $\overline{r} = p(x)$. Mais en pratique, le calcul de $\widehat{c}$ est effectué en arithmétique flottante, et sera donc également entaché d'erreurs d'arrondi. Cette compensation permet néanmoins d'améliorer significativement la précision du résultat calculé, comme cela est prouvé dans la section suivante.

Les performances de l'algorithme 4.3 seront étudiées en détail au chapitre 5. Nous pouvons toutefois déjà noter que CompHorner requiert $22n + \mathcal{O}(1)$ opérations flottantes.

4.3 Précision du schéma de Horner compensé

Nous montrons dans cette section que l'algorithme 4.3 admet une borne d'erreur bien plus précise que celle de l'algorithme classique : CompHorner fournit en effet un résultat aussi précis que si les calculs avaient été effectués avec le schéma de Horner en précision de travail doublée.

Théorème 4.4. *On considère un polynôme p à coefficients flottants de degré n et x un nombre flottant. On suppose $2n\mathbf{u} < 1$. Alors, en l'absence d'underflow,*

$$|\mathsf{CompHorner}(p, x) - p(x)| \leq \mathbf{u}|p(x)| + \gamma_{2n}^2 \widetilde{p}(x). \tag{4.9}$$

Avant de démontrer ce résultat, nous formulons ci-dessous deux lemmes techniques. Le lemme 4.5 permet de borner l'erreur directe commise en évaluant la somme de deux polynômes par la schéma de Horner. La preuve que nous en donnons est succincte car l'analyse d'erreur se déduit facilement de celle du schéma de Horner effectuée au chapitre 2, Section 2.4.

Lemme 4.5. *Soient $p(x) = \sum_{i=0}^{n} a_i x^i$ et $q(x) = \sum_{i=0}^{n} b_i x^i$ deux polynômes à coefficients flottants de degré n, et soit x un flottant. On considère l'évaluation flottante de $(p+q)(x)$ calculée par $\mathsf{Horner}(p \oplus q, x)$. Alors, en l'absence d'underflow, on a*

$$|(p+q)(x) - \mathsf{Horner}(p \oplus q, x)| \leq \gamma_{2n+1}(\widetilde{p+q})(x), \tag{4.10}$$

avec $(\widetilde{p+q})(x) := \sum_{i=0}^{n} |a_i + b_i||x|^i$.

Preuve. En procédant de la même manière que pour l'analyse d'erreur de l'évaluation d'un polynôme à coefficients flottants par le schéma de Horner (voir Section 2.4), on obtient :

$$\mathsf{Horner}(p \oplus q, x) = (1+\theta_{2n+1})(a_n + b_n)x^n + \sum_{i=0}^{n-1} \big(1+\theta_{2(i+1)}\big)(a_i + b_i)x^i,$$

où chacun des θ_k satisfait $|\theta_k| \leq \gamma_k$. On déduit de cette égalité le résultat annoncé. ■

Le lemme ci-dessous fournit une borne sur l'erreur directe $|\widehat{c} - c|$ entachant le terme correctif calculé $\widehat{c}$. L'intérêt de cette borne est qu'elle fait intervenir le conditionnement absolu $\widetilde{p}(x)$ de d'évaluation de $p(x)$.

Lemme 4.6. *Sous les hypothèses du théorème 4.4, et avec les notations de l'algorithme 4.3,*

$$|\widehat{c} - c| \leq \gamma_{2n-1}\gamma_{2n}\widetilde{p}(x). \tag{4.11}$$

Preuve. Comme $\widehat{c} = \mathsf{Horner}(p_\pi \oplus p_\sigma, x)$ et $c = (p_\pi + p_\sigma)(x)$, avec p_π et p_σ deux polynômes de degré $n-1$ à coefficients flottants, le lemme 4.5 donne

$$|\widehat{c} - c| \leq \gamma_{2n-1}(\widetilde{p_\pi + p_\sigma})(x),$$

et en utilisant la relation (4.6) il vient $|\widehat{c} - c| \leq \gamma_{2n-1}\gamma_{2n}\widetilde{p}(x)$. ■

Nous en venons maintenant à la preuve du théorème 4.4

Preuve du théorème 4.4. L'erreur absolue générée par l'algorithme 4.3 est

$$\begin{aligned} |\overline{r} - p(x)| &= |(\widehat{r} \oplus \widehat{c}) - p(x)|\,, \\ &= |(1+\varepsilon)(\widehat{r} + \widehat{c}) - p(x)|\,, \end{aligned}$$

avec $|\varepsilon| \leq \mathbf{u}$. D'après la relation (4.5), et comme $c = (p_\pi + p_\sigma)(x)$, on a $\widehat{r} = \mathsf{Horner}(p, x) = p(x) - c$. D'où

$$|\overline{r} - p(x)| = |(1+\varepsilon)\,(p(x) - c + \widehat{c}) - p(x)|\,,$$

et

$$|\overline{r} - p(x)| \leq \mathbf{u}|p(x)| + (1+\mathbf{u})|\widehat{c} - c|. \tag{4.12}$$

Nous sommes donc ramenés à majorer l'erreur directe $|\widehat{c} - c|$ du terme correctif calculé. En utilisant le lemme 4.6, on obtient

$$|\overline{r} - p(x)| \leq \mathbf{u}|p(x)| + (1+\mathbf{u})\gamma_{2n-1}\gamma_{2n}\widetilde{p}(x).$$

Il suffit alors de remarquer que $(1+\mathbf{u})\gamma_{2n-1} \leq \gamma_{2n}$ pour obtenir la borne d'erreur (4.9). ■

Il est important d'interpréter le théorème 4.2 en fonction du conditionnement $\mathrm{cond}(p,x)$. En combinant la borne d'erreur (4.9) avec l'expression du conditionnement (2.9) on obtient le corollaire suivant.

Corollaire 4.7. *Avec les mêmes hypothèses que dans le théorème 4.4, l'erreur relative entachant le résultat calculé par le schéma de Horner compensé vérifie*

$$\frac{|\mathsf{CompHorner}(p,x)-p(x)|}{|p(x)|} \leq \mathbf{u} + \gamma_{2n}^2\,\mathrm{cond}(p,x). \tag{4.13}$$

La borne obtenue sur l'erreur relative entachant le résultat compensé est composé de deux termes.

- Le terme $\gamma_{2n}^2\,\mathrm{cond}(p,x) \approx 4n^2\mathbf{u}^2$ montre que le résultat est aussi précis que si les calculs avaient été effectués en précision doublée, c'est à dire en précision $\mathbf{u}^2$.
- Le terme $\mathbf{u}$ indique simplement que le résultat est finalement arrondi vers la précision de travail $\mathbf{u}$.

En particulier, aussi longtemps que $\mathrm{cond}(p,x)$ est petit devant γ_{2n}^{-1}, le terme $\gamma_{2n}^2\,\mathrm{cond}(p,x)$ est négligeable devant $\mathbf{u}$, et la précision du résultat compensé est de l'ordre de l'unité d'arrondi. En conclusion, le corollaire 4.7 nous permet d'affirmer que le résultat fourni par le schéma de Horner compensé est aussi précis que s'il avait été obtenu par le schéma de Horner classique évalué en précision de travail doublée, avec un arrondi final vers la précision courante.

4.4 Arrondi fidèle avec le schéma de Horner compensé

Nous avons vu que la précision du résultat calculé par CompHorner est de l'ordre de l'unité d'arrondi $\mathbf{u}$, tant que le conditionnement est petit devant γ_{2n}^{-1}. Dans cette section, nous précisons cette observation en proposant une condition suffisante sur $\mathrm{cond}(p,x)$ qui assure que le résultat calculé par le schéma de Horner compensé est un arrondi fidèle du résultat exact $p(x)$.

Il est clair que la borne d'erreur (4.9) n'est pas assez fine pour nous aider à trouver une condition suffisante assurant l'arrondi fidèle de l'évaluation. Nous utiliserons par la suite le lemme suivant, issu de [86].

Lemme 4.8 ([86])**.** *Soient r,δ deux réels et soit $\overline{r}$ l'arrondi au plus proche de r. On suppose ici que $\overline{r}$ est un nombre flottant normalisé. Si $|\delta| < \frac{\mathbf{u}}{2}|\overline{r}|$ alors $\overline{r}$ est un arrondi fidèle de $r+\delta$.*

En appliquant le lemme 4.8 au cas du schéma de Horner compensé, on obtient une condition suffisante pour montrer que le résultat compensé est un arrondi fidèle.

Lemme 4.9. *Soit p un polynôme à coefficients flottants, et soit x un flottant. On considère l'approximation $\overline{r} \in \mathbf{F}$ de $p(x)$ calculé avec $\mathsf{CompHorner}(p,x)$, et l'on suppose qu'aucun underflow n'intervient au cours des calculs. On définit $c = (p_\pi + p_\sigma)(x)$. Si $|\widehat{c}-c| < \frac{\mathbf{u}}{2}|\overline{r}|$, alors $\overline{r}$ est un arrondi fidèle de $p(x)$.*

Preuve. On suppose que $|\widehat{c}-c| < \frac{\mathbf{u}}{2}|\overline{r}|$. D'après les notations de l'algorithme 4.3, rappelons que $\overline{r} = \widehat{r}\oplus\widehat{c}$. D'après le lemme 4.8, on en déduit que $\overline{r}$ est un arrondi fidèle de $\widehat{r}+\widehat{c}+c-\widehat{c} = \widehat{r}+c$. Comme $[\widehat{r},p_\pi,p_\sigma] = \mathsf{EFTHorner}(p,x)$, le théorème 4.2 donne $p(x) = \widehat{r}+c$. Donc, $\overline{r}$ est un arrondi fidèle de $p(x)$. ■

Le critère proposé dans le lemme 4.9 porte sur l'erreur $|\widehat{c} - c|$ entachant le terme correctif calculé $\widehat{c}$. La relation (4.11) indique que l'erreur $|\widehat{c} - c|$ est bornée par $\gamma_{2n}^2 \widetilde{p}(x)$. Cette remarque nous permet de proposer un critère basé uniquement sur $\mathrm{cond}(p, x)$, pour assurer que le résultat calculé par CompHorner est un arrondi fidèle.

Théorème 4.10. *Soit p un polynôme à coefficients flottants, et soit x un flottant. On suppose $2n\mathbf{u} < 1$. En l'absence d'underflow, si*

$$\mathrm{cond}(p, x) < \frac{1 - \mathbf{u}}{2 + \mathbf{u}} \frac{\mathbf{u}}{\gamma_{2n}^2}. \tag{4.14}$$

alors CompHorner(p, x) *est un arrondi fidèle du résultat exact $p(x)$.*

La borne sur le conditionnement proposée dans le théorème 4.10 pour garantir l'arrondi fidèle n'est pas facilement interprétable en l'état. En ne s'intéressant qu'aux ordres de grandeurs, on a $(1 - \mathbf{u})/(2 + \mathbf{u}) \approx 1/2$ et $\gamma_{2n}^2 \approx 4n^2\mathbf{u}^2$, d'où

$$\frac{1 - \mathbf{u}}{2 + \mathbf{u}} \frac{\mathbf{u}}{\gamma_{2n}^2} \approx \frac{1}{8n^2} \mathbf{u}^{-1}.$$

On peut donc interpréter le théorème 4.10 de la façon suivante : tant que $\mathrm{cond}(p, x)$ est suffisamment petit devant $\mathbf{u}^{-1}$, le résultat compensé calculé par CompHorner est un arrondi fidèle de $p(x)$.

Pour la preuve du théorème 4.10, nous utiliserons le lemme suivant.

Lemme 4.11. *Sous les hypothèses du théorème 4.10,* CompHorner(p, x) *et $p(x)$ sont de même signe, et*

$$(1 - \mathbf{u})|p(x)| - \gamma_{2n}^2 \widetilde{p}(x) \leq |\mathsf{CompHorner}(p, x)|. \tag{4.15}$$

Preuve. Commençons par montrer que $\overline{r}$ et $p(x)$ sont de même signe si l'inégalité (4.14) est satisfaite. En effet, d'après le théorème 4.4,

$$\left| \frac{\overline{r}}{p(x)} - 1 \right| \leq \mathbf{u} + \gamma_{2n}^2 \,\mathrm{cond}(p, x),$$

d'où, en utilisant l'inégalité (4.14),

$$\frac{\overline{r}}{p(x)} \geq 1 - \left(\mathbf{u} + \gamma_{2n}^2 \,\mathrm{cond}(p, x)\right) > 1 - \left(\mathbf{u} + \frac{1 - \mathbf{u}}{2 + \mathbf{u}}\mathbf{u}\right) = 1 - \frac{3\mathbf{u}}{2 + \mathbf{u}} > 0.$$

Pour ce qui est de la relation (4.15), il suffit l'utiliser l'inégalité (4.9) : si $p(x) > 0$, on a

$$p(x) - \mathbf{u}|p(x)| - \gamma_{2n}^2 \widetilde{p}(x) \leq \overline{r},$$

et si $p(x) < 0$,

$$\overline{r} \leq p(x) + \mathbf{u}|p(x)| + \gamma_{2n}^2 \widetilde{p}(x).$$

Dans les deux cas, en tenant compte du fait que $\overline{r}$ et $p(x)$ sont de même signe, on obtient bien la relation (4.15). ■

Preuve du théorème 4.10. Nous allons montrer que l'on a

$$|\widehat{c} - c| < \frac{\mathbf{u}}{2}\overline{r}.$$

Ainsi le lemme 4.9 nous permettra de conclure que $\overline{r}$ est un arrondi fidèle de $p(x)$.

D'après le lemme 4.6, on sait que $|\widehat{c} - c| \leq \gamma_{2n}^2 \widetilde{p}(x)$, et en utilisant (4.15) on obtient

$$\frac{\mathbf{u}}{2}\left[(1-\mathbf{u})|p(x)| - \gamma_{2n}^2\widetilde{p}(x)\right] \leq \frac{\mathbf{u}}{2}|\overline{r}|.$$

Or,

$$\begin{aligned}
\gamma_{2n}^2\widetilde{p}(x) < \frac{\mathbf{u}}{2}\left[(1-\mathbf{u})|p(x)| - \gamma_{2n}^2\widetilde{p}(x)\right] &\Leftrightarrow (2+\mathbf{u})\gamma_{2n}^2\widetilde{p}(x) < (1-\mathbf{u})|p(x)| \\
&\Leftrightarrow \frac{\widetilde{p}(x)}{|p(x)|} < \frac{1-\mathbf{u}}{2+\mathbf{u}}\frac{\mathbf{u}}{\gamma_{2n}^2} \\
&\Leftrightarrow \operatorname{cond}(p,x) < \frac{1-\mathbf{u}}{2+\mathbf{u}}\frac{\mathbf{u}}{\gamma_{2n}^2}.
\end{aligned}$$

Sous les hypothèses du théorème on écrit alors

$$|\widehat{c} - c| \leq \gamma_{2n}^2\widetilde{p}(x) < \frac{\mathbf{u}}{2}\left[(1-\mathbf{u})|p(x)| - \gamma_{2n}^2\widetilde{p}(x)\right] \leq \frac{\mathbf{u}}{2}|\overline{r}|,$$

donc $\overline{r}$ est bien un arrondi fidèle de $p(x)$. ■

4.5 Expériences numériques

Nous décrivons ici nos expériences numériques concernant le schéma de Horner compensé. Nous présentons dans un premier temps la méthode utilisée pour générer des polynômes arbitrairement mal conditionnés. Ces polynômes nous serviront ensuite à illustrer le lien entre la précision du résultat calculé par le schéma de Horner compensé et le conditionnement.

4.5.1 Génération de polynômes arbitrairement mal conditionnés

La méthode que nous avons utilisée pour générer des polynômes arbitrairement mal conditionnés est très similaire à celle décrite dans [70] dans le cas des produits scalaires. Étant donné le degré d du polynôme à générer, la valeur de l'argument x, une valeur v souhaitée de $p(x)$, et un conditionnement attendu c_{exp}, cette méthode permet de générer un polynôme p tel que $p(x) \approx v$ et $\operatorname{cond}(p,x) \approx c_{exp}$.

Nous présentons ci-dessous cet algorithme de génération sous la forme d'un code Matlab exécutable. Le seul pré-requis est de disposer d'une fonction AccHorner permettant d'évaluer $p(x)$ avec une erreur relative de l'ordre de l'unité d'arrondi. Une telle fonction peut être facilement réalisée sous Matlab à l'aide de la Symbolic Toolbox. Mais pour des raisons de performances, nous avons préféré implanter notre fonction AccHorner à l'aide d'un appel externe à un code C basé sur la bibliothèque MPFR [31].

```
function [p, vact, cact] = GenPoly(d, x, v, cexp)
% GenPoly : Génération de polynômes mal conditionnés
%  entrée  d    degré du polynôme généré
%          x    argument du polynôme généré
%          v    valeur approchée souhaitée de p(x)
%          cexp nombre de conditionnement souhaité
%  sortie  p    vecteur des coefficients
%          vact évaluation précise de p(x)
%          cact valeur de cond(p, x) obtenue
%
%  Les coefficients de p sont stockés selon les
%  conventions de Matlab : p(i) désigne le coefficient
%  de degré d-i+1 du polynôme p.
%

  p = zeros(1, d+1);
  d2 = ceil((d+1)/2);
  b = log2(cexp*abs(v));
  perm = randperm(d+1);

  % génère les d2 premiers coefficients
  p(perm(1)) = (2*rand-1) / x^(d-perm(1)+1);
  p(perm(2)) = (2*rand-1)*2^b / x^(d-perm(2)+1);
  for i=3:d2
    p(perm(i)) = (2*rand-1)*2^(b*rand) / x^(d-perm(i)+1);
  end

  % génère les d+1-d2 coefficients restants
  log2v = log2(abs(v));
  m = [(2*rand(1,d-d2)-1).*(2.^linspace(b,log2v,d-d2))];
  for i=d2+1:d
    p(perm(i)) = (m(i-d2)-AccHorner(p, x)) / x^(d-perm(i)+1);
  end
  p(perm(d+1)) = (v-AccHorner(p, x)) / x^(d-perm(d+1)+1);

  % calcul une évaluation précise de p(x) et de cond(p, x)
  vact = AccHorner(p, x);
  cact = polyval(abs(p), abs(x)) / abs(vact);
```

Supposons que l'on veuille générer un polynôme p, de degré d, tel que $\mathrm{cond}(p, x)$ soit de l'ordre de grandeur de c_{exp}. On choisit *a priori* l'argument x du polynôme, ainsi que la valeur souhaitée v de $p(x)$.

D'après la définition de $\mathrm{cond}(p, x)$, reste alors à déterminer les coefficients de p de manière à ce que $\widetilde{p}(x) \approx c_{exp} \times |v|$ et $p(x) \approx v$. Comme l'argument x est choisi *a priori*, il est important de remarquer que le fait de fixer l'un des coefficients du polynôme revient à fixer la valeur du monôme correspondant à ce coefficient. On procède comme suit.

1. On commence par générer aléatoirement $d_2 = \lfloor n/2 \rfloor$ des monômes de p de manière

à ce que leurs exposants soient compris entre 0 et l'exposant de $c_{exp} \times |v|$. Il est important d'assurer qu'au moins l'un des monômes générés soit du même ordre de grandeur que $c_{exp} \times |v|$. Comme $\widetilde{p}(x)$ est par définition égal à la somme des valeurs absolues des monômes de $p(x)$, ce choix assure $\widetilde{p}(x) \approx c_{exp} \times |v|$.

2. On génère ensuite les $d + 1 - d_2$ monômes restants de manière à ce que $p(x) \approx v$.

Remarquons qu'il est également possible de générer la seconde partie des coefficients en utilisant la boucle suivante :

```
for i=d2+1:d+1
  p(perm(i)) = (v-AccHorner(p, x)) / x^(d-perm(i)+1);
end
```

Cette méthode assure également $p(x) \approx v$, mais provoque l'apparition de nombreux coefficients nuls dans le polynôme généré, ce que nous avons voulu éviter.

4.5.2 Résultats expérimentaux

Sur la figure 4.1, nous comparons les algorithmes Horner et CompHorner, tous deux exécutés sous Matlab en double précision IEEE-754. Pour ces tests nous avons généré un jeu de 700 polynômes de degré 50, dont le nombre de conditionnement varie entre 10^2 et 10^{35}. Les paramètres x et v du générateur ont été chacun choisis aléatoirement dans $[-2^{16}, -2^{-16}] \cup [2^{-16}, 2^{16}]$, de manière à éviter les dépassements de capacités, aussi bien vers l'infini que vers 0. Pour chaque évaluation effectuée, nous reportons la précision relative $|\widehat{y} - p(x)|/|p(x)|$ du résultat calculé $\widehat{y}$ en fonction du nombre de conditionnement du polynôme. Nous représentons également sur la figure 4.1 les bornes d'erreur relative (2.10) et (4.13), pour les résultats calculés respectivement par Horner et CompHorner.

On observe que le schéma de Horner compensé présente bien le comportement attendu. Tant que le conditionnement est inférieur à $\mathbf{u}^{-1}$, l'erreur relative entachant le résultat compensé est plus petite que l'unité d'arrondi $\mathbf{u}$. Ensuite, pour un conditionnement compris entre $\mathbf{u}^{-1}$ et $\mathbf{u}^{-2}$, la précision se dégrade progressivement, jusqu'à ce que l'on ait perdu toute précision pour un conditionnement plus grand que $\mathbf{u}^{-2}$.

On constate néanmoins que les bornes d'erreur relative (2.10) et (4.13) sont toutes les deux pessimistes de plusieurs ordres de grandeur. Ces bornes d'erreur sont en effet des bornes d'erreur *a priori*, établies en effectuant une analyse d'erreur dans le pire cas. Ce phénomène est bien connu, et notamment décrit par Higham : voir [39, p.48, p.65], ainsi que les expériences numériques à propos du schéma de Horner [39, chap.5]). Le caractère pessimiste des bornes d'erreur obtenues dans [70], pour la sommation et le produit scalaire compensés, est également souligné par les auteurs de cet article. Nous reviendrons plus longuement sur cette observation au chapitre 7, où nous verrons comment valider *a posteriori* le résultat compensé calculé par CompHorner.

4.6 Prise en compte de l'underflow

Dans les sections précédentes de ce chapitre, nous avons toujours supposé qu'aucun résultat dénormalisé n'apparaissait au cours des calculs. En particulier, nous avons toujours utilisé le modèle standard (2.5) pour borner l'erreur commise à chaque opération flottante. Ici, nous proposons une borne d'erreur *a priori* pour le schéma de Horner compensé, vérifiée également en présence d'underflow.

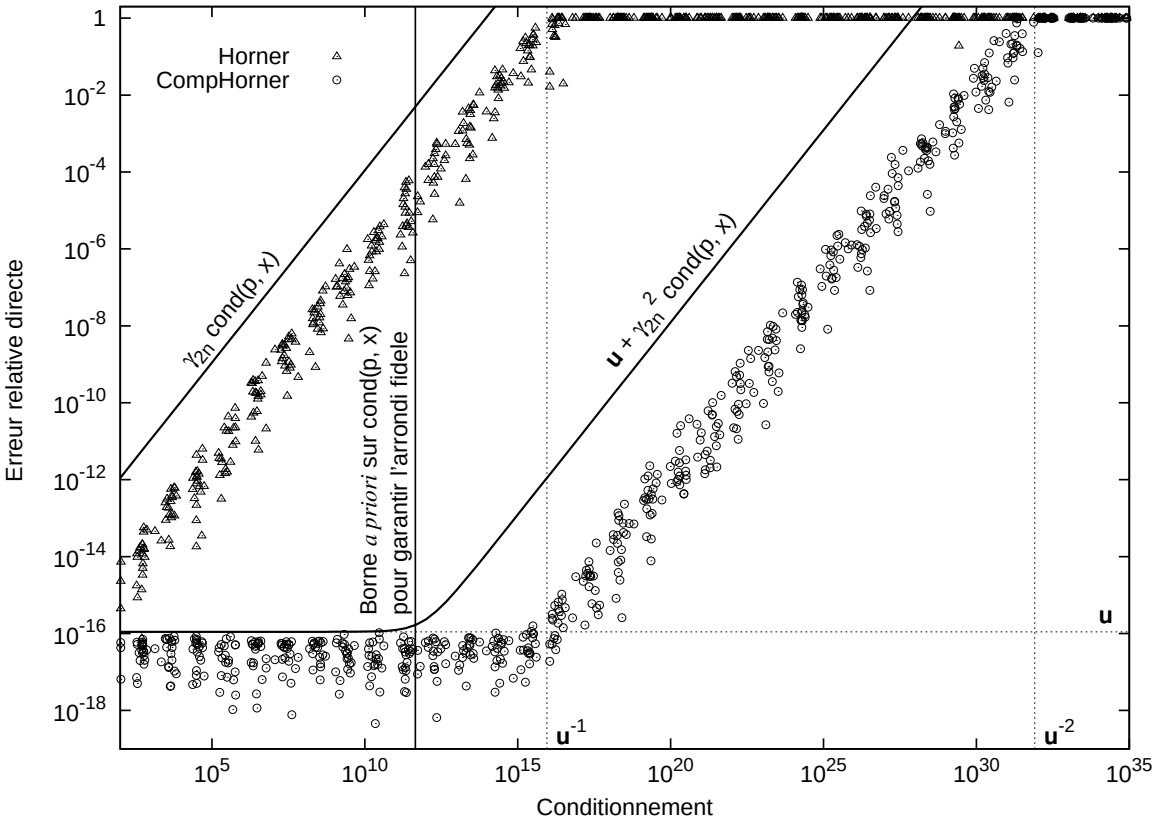


FIG. 4.1 – Précision relative du schéma de Horner et de CompHorner en double précision IEEE-754, avec des polynômes de degré 50.

4.6.1 Résultat

Nous énonçons ci-dessous le théorème 4.12, qui constitue le principal résultat de cette section. Le reste de la section est consacré à la preuve de ce théorème, preuve qui consiste essentiellement à reprendre celle du théorème 4.4, tout en tenant compte d'un possible underflow pour chaque multiplication flottante effectuée.

Rappelons que l'on travaille dans le mode d'arrondi au plus proche. Le plus petit flottant positif normalisé est noté λ, et $\mathbf{v} = 2\lambda\mathbf{u}$ désigne l'unité d'underflow, c'est à dire la distance séparant deux dénormalisés consécutifs.

Théorème 4.12. *On considère un polynôme p à coefficients flottants de degré n et x un nombre flottant. On suppose $n \geq 1$ et $32n\mathbf{u} \leq 1$. Alors, même en présence d'underflow,*

$$|\mathsf{CompHorner}(p, x) - p(x)| \leq \mathbf{u}|p(x)| + \gamma_{2n}^2 \widetilde{p}(x) + 4\mathbf{v} \sum_{i=0}^{n-1} |x|^i. \tag{4.16}$$

Notons que les termes $\mathbf{u}|p(x)|$ et $\gamma_{2n}^2 \widetilde{p}(x)$ apparaissaient déjà dans la borne d'erreur (4.9) valable uniquement en l'absence d'underflow. Le premier borne l'erreur d'arrondi commise lors du calcul du résultat compensé $\overline{r} = \widehat{r} \oplus \widehat{c}$. Le second tient compte des erreurs d'arrondis générées lors du calcul du terme correctif $\widehat{c} = \mathsf{Horner}(p_\pi \oplus p_\sigma, x)$.

Le troisième terme $4\mathbf{v} \sum_{i=0}^{n-1} |x|^i$ dans la borne (4.16) prend désormais en compte la possibilité d'underflow pour chacun des produits effectués par le schéma de Horner compensé.

Néanmoins, cette borne d'erreur n'est pas facile à interpréter : en particulier, il n'est pas facile de voir dans quels cas le terme $4\mathbf{v}\sum_{i=0}^{n-1}|x|^i$ peut être considéré comme négligeable devant $\mathbf{u}|p(x)|$ et $\gamma_{2n}^2\widetilde{p}(x)$.

Comme nous l'avons déjà vu à la Section 4.3, tant que $\mathrm{cond}(p,x)$ est petit devant $\mathbf{u}^{-1}$, on a $\gamma_{2n}^2\widetilde{p}(x) \lesssim \mathbf{u}|p(x)|$. Plus formellement,

$$\mathrm{cond}(p,x) \leq \frac{\mathbf{u}}{\gamma_{2n}^2} \implies \gamma_{2n}^2\widetilde{p}(x) \leq \mathbf{u}|p(x)|.$$

Pour essayer d'interpréter le théorème 4.12, nous présentons ci-dessous deux situations dans lesquelles

$$4\mathbf{v}\sum_{i=0}^{n-1}|x|^i \leq \gamma_{2n}^2\widetilde{p}(x). \tag{4.17}$$

Lorsque l'inégalité (4.17) est satisfaite, nous pouvons considérer que l'effet de l'underflow sur la qualité du résultat compensé n'est pas plus important que celui du conditionnement.

Commençons par déduire une condition suffisante pour que l'inégalité (4.17) soit satisfaite. Comme $2n\mathbf{u} \leq \gamma_{2n}$, on a

$$\frac{4\mathbf{v}}{\gamma_{2n}^2} \leq \frac{4\mathbf{v}}{4n^2\mathbf{u}^2} = \frac{2\lambda}{n^2\mathbf{u}}.$$

On obtient donc la condition suffisante suivante,

$$\frac{2\lambda}{n^2\mathbf{u}}\frac{\sum_{i=0}^{n-1}|x|^i}{\widetilde{p}(x)} \leq 1 \implies 4\mathbf{v}\sum_{i=0}^{n-1}|x|^i \leq \gamma_{2n}^2\widetilde{p}(x). \tag{4.18}$$

Notons bien que $2\lambda\mathbf{u}^{-1} \ll 1$ dans le cadre d'une arithmétique raisonnable, en particulier celui de la norme IEEE-754. On supposera donc

$$\frac{2\lambda}{n^2\mathbf{u}} \leq 1$$

par la suite. En utilisant (4.18), il est maintenant possible d'isoler deux cas particuliers.

Premièrement, lorsque tous les coefficients du polynôme p sont de valeur absolue plus grande que 1, c'est à dire $|a_i| \geq 1$ pour $i = 1:n$, on a

$$\frac{\sum_{i=0}^{n-1}|x|^i}{\widetilde{p}(x)} = \frac{\sum_{i=0}^{n-1}|x|^i}{\sum_{i=0}^{n}|a_n||x|^i} \leq 1.$$

D'après (4.18), les effets de l'underflow restent donc négligeables dans ce cas, au sens de l'inégalité (4.17).

Deuxièmement, le cas $|x| \leq 1$: en remarquant que $|p(x)| \leq \widetilde{p}(x)$, et en majorant simplement le numérateur par n, on a

$$\frac{\sum_{i=0}^{n-1}|x|^i}{\widetilde{p}(x)} \leq \frac{n}{|p(x)|}.$$

La condition suffisante (4.18) s'écrit donc maintenant,

$$\frac{2\lambda}{n\mathbf{u}} \leq |p(x)| \implies 4\mathbf{v}\sum_{i=0}^{n-1}|x|^i \leq \gamma_{2n}^2\widetilde{p}(x).$$

On retiendra donc que dans le cas $|x| \leq 1$, l'effet de l'underflow reste négligeable tant que $|p(x)|$ est suffisamment grand devant λ.

Dans les autres cas, en particulier lorsque les coefficients du polynôme p sont en valeur absolue petits devant 1, ou lorsque $|x| > 1$, on ne peut rien conclure en toute généralité.

4.6.2 Évaluation de la somme de deux polynômes

Nous démontrons ici le lemme 4.13, qui permet de majorer l'erreur commise lors du calcul du terme correctif. Rappelons que le calcul du terme correctif consiste en l'évaluation de la somme de deux polynômes à coefficients flottants par le schéma de Horner.

Soient $p(x) = \sum_{i=0}^{n} a_i x^i$ et $q(x) = \sum_{i=0}^{n} b_i x^i$ deux polynômes à coefficients flottants, et soit x un argument flottant. On définit les réels r_i par

$$r_n = a_n + b_n \quad \text{et} \quad r_i = r_{i+1}x + (a_i + b_i), \quad \text{pour} \quad i = n-1, \ldots, 0.$$

Définissons les $\widehat{r}_i$ comme l'évaluation en arithmétique flottante des r_i, c'est à dire

$$\widehat{r}_n = a_n \oplus b_n \quad \text{et} \quad \widehat{r}_i = \widehat{r}_{i+1} \otimes x \oplus (a_i \oplus b_i) \quad \text{pour} \quad i = n-1, \ldots, 0.$$

On a en particulier $r_0 = (p+q)(x)$ et $\widehat{r}_0 = \mathsf{Horner}(p \oplus q, x)$.

Lemme 4.13. *Même en présence d'underflow, on a*

$$|\mathsf{Horner}(p \oplus q, x) - (p+q)(x)| \leq \gamma_{2n+1}(\widetilde{p+q})(x) + (1+\gamma_{2n-1})\mathbf{v}\sum_{i=0}^{n-1}|x|^i.$$

Les compteurs $\langle k \rangle$ introduits par Stewart [39, p.68] sont à nouveau utilisés ici pour l'analyse d'erreurs. En présence d'underflow, ils seront manipulés comme indiqué ci-après. Si $\circ \in \{+, -, \times, /\}$ est une opération élémentaire, alors son évaluation flottante pour $a, b \in \mathbf{F}$ vérifie,

$$\mathrm{fl}(a \circ b) = \langle 1 \rangle (a \circ b) + \eta,$$

avec $\eta = 0$ si $\circ \in \{+, -\}$, et $|\eta| \leq \mathbf{v}$ sinon.

Preuve du lemme 4.13. On a $\widehat{r}_n = a_n \oplus b_n = \langle 1 \rangle (a_n + b_n)$, et, pour $i = n-1, \ldots, 0$,

$$\widehat{r}_i = \langle 2 \rangle \widehat{r}_{i+1} x + \langle 2 \rangle (a_i + b_i) + \langle 1 \rangle \eta_i,$$

avec $|\eta_i| \leq \mathbf{v}$. On peut ainsi montrer que

$$\widehat{r}_0 = \langle 2n+1 \rangle (a_n + b_n)x^n + \sum_{i=0}^{n-1} \langle 2i+2 \rangle (a_i + b_i)x^i + \sum_{i=0}^{n-1} \langle 2i+1 \rangle \eta_i x^i.$$

Comme chacun des compteurs d'erreur $\langle k \rangle$ désigne un facteur $(1+\theta_k)$ avec $|\theta_k| \leq \gamma_k$, on déduit facilement de l'équation précédente le résultat annoncé. ■

4.6.3 Preuve du théorème 4.12

Si le résultat de l'un des produits effectués au cours de l'exécution de $\mathsf{EFTHorner}(p, x)$ (algorithme 4.1) provoque un underflow, on n'obtient plus une transformation exacte pour $p(x)$. Néanmoins, le théorème suivant nous permet encore d'écrire une relation exacte entre $p(x)$ et les sorties de $\mathsf{EFTHorner}(p, x)$.

Théorème 4.14. *Soit $p(x) = \sum_{i=0}^{n} a_i x^i$ un polynôme de degré n à coefficients flottants, et soit x un nombre flottant. On suppose $\gamma_{2n} \leq 1$. L'algorithme 4.1 calcule à la fois $\mathsf{Horner}(p, x)$ et deux polynômes p_π et p_σ de degré $n-1$ à coefficients flottants, tels que*

$$p(x) = \mathsf{Horner}(p, x) + (p_\pi + p_\sigma)(x) + 5\sum_{i=0}^{n-1} \eta_i x^i, \tag{4.19}$$

avec $|\eta_i| \leq \mathbf{v}$. De plus,

$$\widetilde{(p_\pi + p_\sigma)}(x) \leq (\widetilde{p_\pi} + \widetilde{p_\sigma})(x) \leq \gamma_{2n}\widetilde{p}(x) + 7\mathbf{v}\sum_{i=0}^{n-1} |x|^i. \tag{4.20}$$

Preuve. D'après le théorème 3.10, la prise en compte des underflows dans $\mathsf{TwoProd}$ s'écrit $r_{i+1}x = p_i + \pi_i + 5\eta_i$, avec $|\eta_i| \leq \mathbf{v}$. D'autre part, rappelons que TwoSum reste une transformation exacte pour l'addition de deux flottants même en présence d'underflow (voir théorème 3.6). On a donc $p_i + a_i = r_i + \sigma_i$. Ainsi, pour $i = 0, \ldots, n-1$, nous avons $r_i = r_{i+1}x + a_i - \pi_i - \sigma_i - 5\eta_i$. Comme $r_n = a_n$, on peut montrer par récurrence qu'à la fin de la boucle,

$$r_0 = \sum_{i=0}^{n} a_i x^i - \sum_{i=0}^{n-1} (\pi_i + \sigma_i) x^i - 5\sum_{i=0}^{n-1} \eta_i x^i,$$

ce qui démontre la relation (4.19).

Afin de prouver la relation (4.20), démontrons maintenant par récurrence que l'on a, pour $i = 1, \ldots, n$,

$$|p_{n-i}| \leq (1+\gamma_{2i-1}) \sum_{j=1}^{i} |a_{n-i+j}||x^j| + (1+\gamma_{2i-2})\mathbf{v}\sum_{j=0}^{i-1} |x^j|, \tag{4.21}$$

$$|r_{n-i}| \leq (1+\gamma_{2i}) \sum_{j=0}^{i} |a_{n-i+j}||x^j| + (1+\gamma_{2i-1})\mathbf{v}\sum_{j=0}^{i-1} |x^j|. \tag{4.22}$$

Pour $i = 1$, comme $r_n = a_n$, on a

$$|p_{n-1}| = |a_n \otimes x| \leq (1+\mathbf{u})|a_n||x| + \mathbf{v} \leq (1+\gamma_1)|a_n||x| + \mathbf{v},$$

donc la relation (4.21) est vraie. De la même façon, nous avons

$$|r_{n-1}| \leq (1+\mathbf{u})\left((1+\gamma_1)|a_n||x| + \mathbf{v} + |a_{n-1}|\right) \leq (1+\gamma_2)\left(|a_n||x| + |a_{n-1}|\right) + (1+\gamma_1)\mathbf{v},$$

donc la relation (4.22) est aussi vérifiée. Supposons maintenant que (4.21) et (4.22) sont vraies pour un entier i tel que $1 \leq i < n$. Puisque $|p_{n-(i+1)}| = |r_{n-i} \otimes x| \leq (1+\mathbf{u})|r_{n-i}||x| + \mathbf{v}$, par hypothèse de récurrence on a

$$|p_{n-(i+1)}| \leq (1+\gamma_{2(i+1)-1}) \sum_{j=1}^{i+1} |a_{n-(i+1)+j}||x^j| + (1+\gamma_{2(i+1)-2})\mathbf{v}\sum_{j=0}^{i} |x^j|.$$

D'autre part, puisque $|r_{n-(i+1)}| = |p_{n-(i+1)} \oplus a_{n-(i+1)}| \leq (1+\mathbf{u})(|p_{n-(i+1)}| + |a_{n-(i+1)}|)$, on déduit que

$$|r_{n-(i+1)}| \leq \left(1+\gamma_{2(i+1)}\right) \sum_{j=0}^{i+1} |a_{n-(i+1)+j}||x^j| + \left(1+\gamma_{2(i+1)-1}\right) \mathbf{v} \sum_{j=0}^{i} |x^j|.$$

Les relations (4.21) and (4.22) sont donc vraies par récurrence. Ainsi, pour $i = 1, \ldots, n$,

$$\begin{aligned} |p_{n-i}||x^{n-i}| &\leq (1+\gamma_{2n-1})\widetilde{p}(x) + (1+\gamma_{2n-2})\mathbf{v} \sum_{j=0}^{n-1} |x^j|, \quad \text{et} \\ |r_{n-i}||x^{n-i}| &\leq (1+\gamma_{2n})\widetilde{p}(x) + (1+\gamma_{2n-1})\mathbf{v} \sum_{j=0}^{n-1} |x^j|. \end{aligned}$$

Puisque $[p_i, \pi_i] = \mathsf{TwoProd}_{xh,xl}(r_{i+1}, x)$ et $[r_i, \sigma_i] = \mathsf{TwoSum}(p_i, a_i)$, d'après les théorèmes 3.10 et 3.6, on a $|\pi_i| \leq \mathbf{u}|p_i| + 5\mathbf{v}$ et $|\sigma_i| \leq \mathbf{u}|r_i|$ pour $i = 0, \ldots, n-1$. Par conséquent,

$$(\widetilde{p}_\pi + \widetilde{p}_\sigma)(x) = \sum_{i=0}^{n-1} (|\pi_i| + |\sigma_i|)|x^i| \leq \mathbf{u} \sum_{i=1}^{n} (|p_{n-i}| + |r_{n-i}|)|x^{n-i}| + 5\mathbf{v} \sum_{i=0}^{n-1} |x^i|.$$

On obtient alors

$$\begin{aligned} (\widetilde{p}_\pi + \widetilde{p}_\sigma)(x) &\leq n\mathbf{u}(2+\gamma_{2n-1}+\gamma_{2n})\widetilde{p}(x) + \left(5 + n\mathbf{u}(2+\gamma_{2n-2}+\gamma_{2n-1})\right)\mathbf{v} \sum_{i=0}^{n-1} |x^i| \\ &\leq 2n\mathbf{u}(1+\gamma_{2n})\widetilde{p}(x) + \left(5 + 2n\mathbf{u}(1+\gamma_{2n})\right)\mathbf{v} \sum_{i=0}^{n-1} |x^i| \\ &\leq \gamma_{2n}\widetilde{p}(x) + (5+\gamma_{2n})\mathbf{v} \sum_{i=0}^{n-1} |x^i|. \end{aligned}$$

Puisque $\gamma_{2n} \leq 1$, cela conclut la preuve du résultat annoncé. ■

Le lemme suivant fournit une borne sur l'erreur directe entachant le terme correctif calculé $\widehat{c}$, valide en présence d'underflow.

Lemme 4.15. *Avec les hypothèses du théorème 4.12, et les notations de l'algorithme 4.3,*

$$|c - \widehat{c}| \leq \gamma_{2n-1}\gamma_{2n}\widetilde{p}(x) + 2\mathbf{v} \sum_{i=0}^{n-1} |x|^i. \tag{4.23}$$

Preuve. D'après le lemme 4.13, comme p_π et p_σ sont deux polynômes degré $n-1$, on a

$$|c - \widehat{c}| \leq \gamma_{2n-1}\widetilde{(p_\pi + p_\sigma)}(x) + (1+\gamma_{2n-3})\mathbf{v} \sum_{i=0}^{n-2} |x|^i.$$

En utilisant la relation (4.20), on obtient

$$\begin{aligned} |c - \widehat{c}| &\leq \gamma_{2n-1}\gamma_{2n}\widetilde{p}(x) + (1+\gamma_{2n-3})\mathbf{v} \sum_{i=0}^{n-2} |x|^i + 6\gamma_{2n-1}\mathbf{v} \sum_{i=0}^{n-1} |x|^i \\ &\leq \gamma_{2n-1}\gamma_{2n}\widetilde{p}(x) + (1+\gamma_{2n-3}+6\gamma_{2n-1})\mathbf{v} \sum_{i=0}^{n-1} |x|^i. \end{aligned}$$

On a $\gamma_{2n-3} + 6\gamma_{2n-1} \leq 8\gamma_{2n} \leq \gamma_{16n}$. Or, $\gamma_{16n} \leq 1$ tant que $32n\mathbf{u} \leq 1$. Par hypothèse on obtient donc bien la relation (4.23). ■

Le lemme 4.15 nous permet maintenant de démontrer le théorème 4.12.

Preuve du théorème 4.12. Comme dans la preuve du théorème 4.4, on a

$$|\overline{r} - p(x)| = |(\widehat{r} \oplus \widehat{c}) - p(x)| = |(1+\varepsilon)(\widehat{r} + \widehat{c}) - p(x)|,$$

avec $|\varepsilon| \leq \mathbf{u}$. D'après la relation (4.19), et comme $c = (p_\pi + p_\sigma)(x)$,

$$\widehat{r} = \mathsf{Horner}(p, x) = p(x) - c - 5\mathbf{v}\sum_{i=0}^{n-1} x^i.$$

D'où

$$|\overline{r} - p(x)| = \left|(1+\varepsilon)\left(p(x) - c - 5\mathbf{v}\sum_{i=0}^{n-1} x^i - \widehat{c}\right) - p(x)\right|,$$

et

$$|\overline{r} - p(x)| \leq \mathbf{u}|p(x)| + (1+\mathbf{u})|\widehat{c} - c| + (1+\mathbf{u})\mathbf{v}\sum_{i=0}^{n-1}|x|^i. \tag{4.24}$$

En utilisant (4.23) pour majorer $|\widehat{c} - c|$, on obtient

$$|\overline{r} - p(x)| \leq \mathbf{u}|p(x)| + (1+\mathbf{u})\gamma_{2n-1}\gamma_{2n}\widetilde{p}(x) + (1+\mathbf{u})3\mathbf{v}\sum_{i=0}^{n-1}|x|^i.$$

En remarquant $(1+\mathbf{u})\gamma_{2n-1} \leq \gamma_{2n}$ et $(1+\mathbf{u})3 \leq 4$, on obtient la borne d'erreur (4.16). ■

4.7 Conclusions

Le schéma de Horner compensé, présenté dans ce chapitre, permet l'évaluation précise de polynômes univariés à coefficients flottants. À l'aide d'une analyse d'erreur *a priori*, nous avons démontré que le résultat compensé calculé par cet algorithme est aussi précis que s'il avait été calculé en précision doublée $\mathbf{u}^2$, avec un arrondi final vers la précision de travail $\mathbf{u}$.

Nous avons également démontré une condition suffisante portant sur le nombre de conditionnement pour assurer l'arrondi fidèle : tant que le conditionnement de l'évaluation polynomiale est suffisamment petit devant $\mathbf{u}^{-1}$, le schéma de Horner compensé calcule un arrondi fidèle du résultat exact.

Le problème de la prise en compte des cas d'underflow reste selon nous un problème ouvert : comme nous l'avons vu, l'analyse d'erreur est dans ce cas particulièrement fastidieuse, et la borne d'erreur obtenue difficilement interprétable. Notons néanmoins que nous sommes parvenu à isoler deux cas dans lesquels la présence d'underflow peut être considérée comme négligeable.

Insistons sur le fait que le schéma de Horner compensé ne nécessite que des opérations flottantes élémentaires (additions et multiplications), effectuées à une précision de travail $\mathbf{u}$ fixée, dans le mode d'arrondi au plus proche. Notre algorithme est donc facilement portable,

en particulier dans tout environnement disposant d'une arithmétique flottante conforme à la norme IEEE-754.

Aux chapitres 5, 6 et 7, nous poursuivrons l'étude du schéma de Horner compensé. Au chapitre suivant, nous montrerons qu'il s'agit d'un algorithme efficace en terme de temps de calcul, notamment face au schéma de Horner en arithmétique double-double qui permet également de simuler une précision de travail doublée.

Chapitre 5

Performances du schéma de Horner compensé

Plan du chapitre : Dans la Section 5.2, nous montrons comment sont implantés en pratiques les algorithmes CompHorner et DDHorner, et nous reportons les performances mesurées pour nos implantations. Nous insistons également sur la principale différence algorithmique entre CompHorner et DDHorner : les étapes de renormalisation nécessaires au calcul avec les double-doubles sont évitées dans le schéma de Horner compensé. La Section 5.3 est dédiée à l'étude du parallélisme d'instructions présent dans les deux algorithmes considérés, et met en évidence le fait que CompHorner exhibe plus de parallélisme d'instructions que DDHorner.

5.1 Introduction

Le but de ce chapitre est de mettre en évidence les bonnes performances en pratique du schéma de Horner compensé. Le doublement de la précision des calculs obtenu à l'aide du schéma de Horner compensé (algorithme CompHorner) introduit nécessairement un surcoût en comparaison du schéma de Horner classique (algorithme Horner). Définissons ce surcoût comme étant égal au ratio du temps d'exécution de CompHorner sur celui de Horner. Il est clair que ce surcoût dépend de l'architecture utilisée pour effectuer la mesure.

Mais il est commun de penser que le surcoût introduit par CompHorner doit être sensiblement égal au ratio du nombre d'opérations flottantes exécutées par CompHorner sur le nombre d'opérations flottantes effectuées par Horner : on devrait alors s'attendre à un surcoût de l'ordre de 11. Or, au travers de mesures effectuées sur un certain nombre d'architectures récentes, nous verrons que ce surcoût est en pratique de l'ordre de 3, ce qui est donc bien inférieur à celui que l'on aurait pu attendre au regard du décompte des opérations flottantes. Nous observons aussi ce phénomène dans le cas du schéma de Horner basé sur les double-doubles (algorithme DDHorner). Néanmoins, nos expériences montrent que CompHorner s'exécute en pratique environ 2 fois plus rapidement que DDHorner.

Dans un second temps, nous avancerons un certain nombre d'éléments afin d'expliquer les bonnes performances du schéma de Horner compensé, en nous basant sur la notion de parallélisme d'instructions (ILP pour *instruction-level parallélism*). Citons à ce sujet Hennessy et Patterson [33, p.172] :

> All processors since about 1985, including those in the embedded space, use pipelining to overlap the execution of instructions and improve performances. This potential overlap among instructions is called instruction-level parallelism since the instructions can be evaluated in parallel.

La technique bien connue du pipeline exploite le parallélisme disponible entre les instructions d'un programme, en permettant le recouvrement de l'exécution de ses instructions lorsqu'elles sont indépendantes. Ce mécanisme permet généralement une exécution plus rapide du programme, notamment plus rapide que si toutes ses instructions étaient exécutées en séquence.

Il est important de noter que la quantité de parallélisme d'instructions disponible dans un programme est une caractéristique intrinsèque de ce programme, indépendante notamment de l'architecture sur laquelle on l'exécute. En particulier, si toutes les instructions d'un code forment une chaîne de dépendance, alors elles doivent nécessairement être exécutées en séquence, ce qui signifie que ce code ne présente pas de parallélisme d'instructions. D'autre part, bien que les processeurs permettent l'exécution en parallèle de nombreuses instructions, les programmes sont pour la plupart toujours écrits, dans les langages comme le C ou le FORTRAN, d'une manière purement séquentielle. De nombreuses techniques, matérielles et logicielles, ont donc été développées afin d'exploiter le parallélisme d'instructions disponible dans les programmes.

Dans ce chapitre, nous proposons une analyse détaillée du parallélisme d'instructions disponible dans l'algorithme de Horner compensée (CompHorner, algorithme 4.3), et dans l'algorithme de Horner basé sur les doubles-doubles (DDHorner, algorithme 3.27). Nous quantifions le nombre moyen d'instructions de chacun de ces algorithmes qui peuvent théoriquement être exécutées simultanément sur un processeur idéal. Ce processeur idéal, défini dans [33, p.240], est un processeur dans lequel toutes les contraintes artificielles sur le parallélisme d'instructions sont supprimées. Dans ce contexte, l'IPC (« instruction per clock ») théorique est environ 6 fois meilleur pour l'algorithme de Horner compensé que pour son équivalent en arithmétique double-double.

Chaque opération arithmétique faisant intervenir un double-double se termine par une étape de renormalisation [81, 57]. Nous montrons également que l'absence de telles étapes de renormalisation dans l'algorithme de Horner compensé explique le fait qu'il présente plus de parallélisme d'instructions. De notre point de vu, cela fournit une explication qualitative des bonnes performances de cet algorithme compensé sur les architectures modernes.

5.2 Performances en pratique

Dans cette section, nous présentons comment sont implantés CompHorner et DDHorner, et montrons quels sont les surcoûts introduits en pratique par ces deux algorithmes.

5.2.1 Implantations pratique de CompHorner

On considère l'évaluation de $p(x)$ à l'aide du schéma de Horner compensé (algorithme 4.3). Tel que présentée au chapitre 4, cette évaluation compensée s'effectue en trois étapes :

1. calcul de $\mathsf{Horner}(p, x)$ et des deux polynômes p_π et p_σ de degré $n-1$ par $\mathsf{EFTHorner}(p, x)$,
2. évaluation de $(p_\pi + p_\sigma)(x)$ avec $\mathsf{Horner}(p_\pi \oplus p_\sigma, x)$,

3. finalement, calcul du résultat compensé $\overline{r}$.

Il est facile de voir que les deux premières étapes peuvent être regroupées, en évitant ainsi le stockage des polynômes p_π et p_σ. Le schéma de Horner compensé sera donc implanté en pratique à l'aide d'une simple boucle, comme indiqué ci-dessous.

function $r = \mathsf{CompHorner}(P, x)$
  $[xh, xl] = \mathsf{Split}(x)$
  $r_n = a_n$; $c_n = 0$
  for $i = n-1 : -1 : 0$
    $[p_i, \pi_i] = \mathsf{TwoProd}_{xh,xl}(r_{i+1}, x)$
    $[r_i, \sigma_i] = \mathsf{TwoSum}(p_i, a_i)$
    $c_i = c_{i+1} \otimes x \oplus (\pi_i \oplus \sigma_i)$
  end
  $r = r_0 \oplus c_0$

Notons bien que le découpage de x en $xh + xl$ n'est effectué qu'une seule fois avant le début de la boucle, de manière à réduire le coût du calcul de $\mathsf{TwoProd}(r_{i+1}, x)$. C'est ce qui est fait dans l'implantation en langage C de CompHorner proposée ci-dessous. On peut voir que cette implantation nécessite $22n + 5$ opérations flottantes.

```
double CompHorner(double *P, unsigned int n, double x) {
  double p, r, c, pi, sig, x_hi, x_lo, hi, lo, t;
  int i;

  /* Split(x_hi, x_lo, x) */
  t = x * _splitter_; x_hi = t - (t - x); x_lo = x - x_hi;

  r = P[n]; c = 0.0;
  for(i=n-1; i>=0; i--) {
    /* TwoProd(p, pi, s, x); */
    p = r * x;
    t = r * _splitter_; hi = t - (t - r); lo = r - hi;
    pi = (((hi*x_hi - p) + hi*x_lo) + lo*x_hi) + lo*x_lo;

    /* TwoSum(s, sigma, p, P[i]); */
    r = p + P[i];
    t = r - p;
    sig = (p - (r - t)) + (P[i] - t);

    /* Computation of the error term */
    c = c * x + (pi+sig);
  }
  return(r+c);
}
```

5.2.2 Algorithme de Horner en arithmétique double-double

Nous rappelons ci-dessous le schéma de Horner effectué en utilisant l'arithmétique double-double — voir chapitre 3, algorithme 3.27. Rappelons également qu'un double-

double **a** est un couple de flottants (ah, al) avec $\mathbf{a} = ah + al$ et $|al| \leq \frac{1}{2}\,\mathrm{ulp}(ah)$. Cette dernière propriété impose une étape de renormalisation à la fin de chaque opération arithmétique sur les double-doubles : c'est ici l'algorithme FastTwoSum (algorithme 3.1) qui assure cette renormalisation. Par exemple, la ligne $[sh_i, sl_i] = \mathsf{FastTwoSum}(th, tl)$ dans l'algorithme 3.27 assure que le couple (sh_i, sl_i) satisfait $|sl_i| \leq \frac{1}{2}\,\mathrm{ulp}(sh_i)$, et constitue donc bien un double-double valide.

Algorithme 5.1. Algorithme de Horner en arithmétique double-double

function $r = \mathsf{DDHorner}(P, x)$
$[xh, xl] = \mathsf{Split}(x)$
$sh_n = a_i$; $sl_n = 0$
for $i = n - 1 : -1 : 0$

% double-double = double-double × double :
% $(ph_i, pl_i) = (sh_{i+1}, sl_{i+1}) \otimes x$
$[th, tl] = \mathsf{TwoProd}_{x_h, x_l}(sh_{i+1}, x)$
$tl = sl_{i+1} \otimes x \oplus tl$
$[ph_i, pl_i] = \mathsf{FastTwoSum}(th, tl)$

% double-double = double-double + double :
% $(sh_i, sl_i) = (ph_i, pl_i) \oplus a_i$
$[th, tl] = \mathsf{TwoSum}(ph_i, a_i)$
$tl = tl \oplus pl_i$
$[sh_i, sl_i] = \mathsf{FastTwoSum}(th, tl)$

end
$r = sh_0$

A nouveau, on constate que x est découpé une fois pour toute avant le début de la boucle. Nous fournissons également ci-dessous une implantation en langage C du schéma de Horner avec l'arithmétique double-doubles. Cette implantation requiert $28n+4$ opérations flottantes.

```
double DDHorner(double *P, unsigned int n, double x) {
  double r_h, r_l, t_h, t_l, x_hi, x_lo, hi, lo, t;
  int i;

  /* Split(x_hi, x_lo, x) */
  t = x * _splitter_; x_hi = t - (t - x); x_lo = x - x_hi;

  r_h = P[n]; r_l = 0.0;
  for(i=n-1; i>=0; i--) {
    /* (r_h, r_l) = (r_h, r_l) * x */
    t = r_h * _splitter_; hi = t - (t - r_h); lo = (r_h - hi);
    t_h = r_h * x;
    t_l = (((hi*x_hi-t_h) + hi*x_lo) + lo*x_hi) + lo*x_lo;
    t_l += r_l * x;
    r_h = t_h + t_l;
```

```
    r_l = (t_h - r_h) + t_l;

    /* (r_h, r_l) = (r_h, r_l) + P[i] */
    t_h = r_h + P[i];
    t = t_h - r_h;
    t_l = ((r_h - (t_h - t)) + (P[i] - t));
    t_l += r_l;
    r_h = t_h + t_l;
    r_l = (t_h - r_h) + t_l;
  }
  return(r_h);
}
```

5.2.3 Résultats expérimentaux

Pour nos expériences, nous avons codé les fonctions C CompHorner et DDHorner, telles que décrites ci-dessus, en utilisant la double précision IEEE-754 comme précision de travail. Les environnements expérimentaux sont listés dans le tableau 5.1.

TAB. 5.1 – Environnements expérimentaux.

Environnement	Description
(I)	Intel Pentium 4, 3.0GHz, GNU Compiler Collection 4.1.2, fpu x87
(II)	Intel Pentium 4, 3.0GHz, GNU Compiler Collection 4.1.2, fpu sse
(III)	Intel Pentium 4, 3.0GHz, Intel C Compiler 9.1, fpu x87
(IV)	Intel Pentium 4, 3.0GHz, Intel C Compiler 9.1, fpu sse
(V)	AMD Athlon 64, 2 GHz, GNU Compiler Collection 4.1.2, fpu sse
(VI)	Itanium 2, 1.5 GHz, GNU Compiler Collection 4.1.1
(VII)	Itanium 2, 1.5 GHz, Intel C Compiler 9.1

Nos mesures sont effectuées à l'aide d'un jeu de 39 polynômes, dont les coefficients sont choisis aléatoirement, et dont le degré varie de 10 à 200 par pas de 5. Pour chaque degré considéré, on mesure le ratio du temps d'exécution de CompHorner sur le temps d'exécution du schéma de Horner classique Horner : cette mesure quantifie le surcoût introduit par CompHorner en comparaison de Horner. On effectue également cette mesure pour l'algorithme DDHorner.

Le tableau Table 5.2 résume les mesures effectuées : nous y avons reporté, pour chaque algorithme et chaque environnement, la moyenne des ratios mesurés, sur l'ensemble des 39 polynômes considérés. Les résultats détaillés de ces mesures sont fournis en annexe.

Premièrement, on observe que les surcoûts mesurés sont toujours significativement plus faibles que ceux auxquels on aurait pu s'attendre d'après le décompte des opérations flottantes. En effet, si l'on ne considère que ces décomptes,

$$\frac{\mathsf{CompHorner}}{\mathsf{Horner}} = \frac{22n+5}{2n} \approx 11, \quad \text{et} \quad \frac{\mathsf{DDHorner}}{\mathsf{Horner}} = \frac{28n+4}{2n} \approx 14.$$

D'autre part, on constate que CompHorner s'exécute au moins deux fois plus rapidement en pratique que DDHorner. Insistons à nouveau sur le fait que les décomptes des opérations

TAB. 5.2 – Surcoût moyen introduit par CompHorner et DDHorner.

Environnement		$\frac{\text{CompHorner}}{\text{Horner}}$	$\frac{\text{DDHorner}}{\text{Horner}}$
(I)	P4, gcc, x87	2.8	8.6
(II)	P4, gcc, sse	3.1	8.9
(III)	P4, icc, x87	2.7	9.0
(IV)	P4, icc, sse	3.3	9.8
(V)	Ath64, gcc, sse	3.2	8.7

flottantes ne permettent pas d'expliquer de telles performances. Dans ce cas, on a en effet

$$\frac{\mathsf{DDHorner}}{\mathsf{CompHorner}} = \frac{28n+4}{22n+5} \approx 1.3.$$

Les simples décomptes d'opérations flottantes sont donc clairement insuffisants pour expliquer les différences de performances entre CompHorner et DDHorner.

5.2.4 Différence algorithmique entre CompHorner et DDHorner

On compare ci-dessous une version modifiée de l'algorithme CompHorner à l'algorithme DDHorner. La modification apportée à CompHorner consiste simplement à changer l'ordre des opérations dans le calcul de c_i en fonction de c_{i+1}, x, π_i et σ_i.

```
function r = CompHorner'(P, x)
  [xh, xl] = Split(x)
  r_n = a_i ; c_n = 0
  for i = n − 1 : −1 : 0

    [a_i, π_i] = TwoProd(r_{i+1}, x)
    t = c_{i+1} ⊗ x ⊕ π_i

    [r_i, σ_i] = TwoSum(p_i, a_i)
    c_i = t_i ⊕ σ_i

  end
  r = r_0 ⊕ c_0
```

```
function r = DDHorner(P, x)
  [xh, xl] = Split(x)
  sh_n = a_i ; sl_n = 0
  for i = n − 1 : −1 : 0
    % (ph_i, pl_i) = (sh_{i+1}, sl_{i+1}) ⊗ x
    [th, tl] = TwoProd_{x_h,x_l}(sh_{i+1}, x)
    tl = sl_{i+1} ⊗ x ⊕ tl
    [ph_i, pl_i] = FastTwoSum(th, tl)

    % (sh_i, sl_i) = (ph_i, pl_i) ⊕ a_i
    [th, tl] = TwoSum(ph_i, a_i)
    tl = tl ⊕ pl_i
    [sh_i, sl_i] = FastTwoSum(th, tl)
  end
  r = sh_0
```

On constate ainsi clairement que les algorithmes CompHorner et DDHorner effectuent essentiellement les mêmes opérations flottantes, à la différence près que les étapes de renormalisation sont supprimées dans CompHorner : CompHorner effectue ainsi au total $6n$ opérations de moins que DDHorner.

La suppression de ces $6n$ opérations flottantes ne permet clairement pas d'expliquer à elle seule le fait que CompHorner s'exécute en pratique deux fois plus rapidement que DDHorner. Cependant nous verrons à la section suivante que, grâce à la suppression de des étapes de renormalisation, CompHorner présente significativement plus de parallélisme d'instruction que DDHorner.

5.3 Comment expliquer les performances du schéma de Horner compensé ?

Le nombre d'instructions qui peuvent être exécutées en parallèle dans un programme donné détermine le parallélisme d'instructions disponible. Si deux instructions sont indépendantes, les deux peuvent s'exécuter simultanément dans un pipeline, en supposant que ce pipeline dispose de suffisant de ressources. Si au contraire une instruction dépend de l'autre, alors ces deux instructions devront s'exécuter en séquence. Les dépendances sont clairement une propriété des programmes. Le fait que deux instructions indépendantes puissent ou non s'exécuter en parallèle sur un processeur donné relève de l'organisation du pipeline, et du nombre de ressources dont il dispose. La clef pour quantifier le parallélisme d'instructions disponible est donc de déterminer dans quel cas une relation de dépendance existe entre deux instructions.

5.3.1 Nature des dépendances entre instructions

On distingue trois types de dépendances : les dépendances de donnée, les dépendances de nom et les dépendances de contrôle. Nous commençons par rappeler ce que sont les dépendances de donnée, qui nous concerneront plus particulièrement par la suite. On dit qu'une instruction β présente une dépendance de donnée par rapport à une instruction α si l'une des deux conditions suivantes est vérifiée :

- l'instruction α produit un résultat qui est utilisé par l'instruction β,
- l'instruction β dépend d'une instruction γ, qui dépend elle-même de α.

La seconde condition signifie simplement qu'une instruction est dépendante d'une autre, s'il existe une chaîne de dépendances entre ces deux instructions.

Définissons maintenant les dépendances de nom : une dépendance de nom existe lorsque deux instructions utilisent le même nom (registre ou emplacement mémoire), mais qu'il n'y a en fait pas de circulation de données entre ces deux instructions. Deux instructions entre lesquelles il existe une dépendance de nom peuvent tout de même s'exécuter en parallèle, pour peu que le nom utilisé dans ces instructions soit changé de manière à supprimer le conflit. Ce changement peut être réalisé soit de manière statique par le compilateur, soit à l'exécution, au travers d'un mécanisme appelé renommage de registre. Il nous suffit pour l'instant de retenir que les dépendances de nom peuvent toujours être annulées : nous n'en tiendrons donc pas compte dans la suite de notre analyse.

Le dernier type de dépendance est la dépendance de contrôle. Une telle dépendance existe lorsque l'exécution d'une instruction β est conditionnée par une instruction de branchement α : on dit alors qu'il existe une dépendance de contrôle de α vers β. Nous ne tiendrons pas compte non plus des dépendances de contrôle dans notre étude des algorithmes CompHorner et DDHorner, en supposant que l'on dispose d'un prédicteur de branchement parfait. Dans ce cas, tout se passe comme si toutes les instructions de branchement étaient supprimées, et le programme résumé à la suite des instructions se trouvant sur le « bon » chemin de contrôle. En l'occurrence, cela revient à dérouler entièrement la boucle principale de ces deux algorithmes.

5.3.2 Quantification du parallélisme d'instructions via l'IPC

L'IPC (« instruction per clock ») d'un programme, exécuté sur un processeur donné, est le nombre moyen d'instructions de ce programme exécutées par cycle. Une manière d'évaluer le parallélisme d'instructions disponible dans un programme donné est de calculer son IPC sur un processeur idéal, c'est à dire un processeur sur lequel toutes les contraintes artificielles sur le parallélisme d'instructions sont supprimées. Sur un tel processeur, les seules limites quant au parallélisme d'instructions sont celles imposées par les chaînes de dépendances de donnée présentes dans le programme. Plus précisément, on supposera que :

- le processeur peut exécuter un nombre illimité d'instructions indépendantes en un cycle d'horloge ;
- le processeur est capable de supprimer toutes les dépendances, sauf bien-sûr les dépendances de donnée : cela signifie que toute instruction peut commencer à s'exécuter au cycle suivant la fin de l'instruction dont elle dépend ;
- les branchements sont parfaitement prédits : en particulier, tous les branchements conditionnels sont correctement pris ;
- les accès à la mémoire sont également parfaits : les lectures et les écritures en mémoire s'effectuent toujours en un seul cycle.

Ces hypothèses signifient en particulier que ce processeur idéal peut exécuter un nombre arbitrairement grand d'instructions en parallèle, et que toute chaîne d'instructions dépendantes verra ses instructions exécutées à des cycles consécutifs. Par la suite, nous parlerons d'IPC idéal d'un programme pour désigner son IPC sur ce processeur idéal.

Remarque. Dans l'étude que nous menons par la suite du parallélisme d'instructions présent dans Horner, CompHorner et DDHorner, nous ne tiendrons compte que des instructions flottantes. En particulier, nous négligerons les instructions entières, les instructions de contrôle de boucle et celles concernant les accès à la mémoire. En effet, sur le processeur idéal défini ci-dessus, la latence de ces instructions se trouve nécessairement masquée par celle des instructions flottantes. Cela signifie que nous ne quantifions pas exactement l'IPC, mais le nombre moyen d'instructions flottantes de ces programmes exécutées par cycle : en réalité, l'IPC idéal est du même ordre de grandeur, mais toujours supérieur aux valeurs reportées.

5.3.3 IPC de Horner

Le cas du schéma de Horner est particulièrement simple, puisque cet algorithme consiste en l'exécution d'une chaîne de $2n$ opérations flottantes dépendantes. En effet, lors de l'itération i,

- le résultat r_{i+1} de l'itération précédente est multiplié par x,
- puis ce produit est ajouté au coefficient p_i pour donner le résultat r_i de l'itération.

Les $2n$ opérations flottantes du schéma de Horner vont donc s'exécuter sur notre processeur idéal en $2n$ cycles, d'où un l'IPC idéal[1] de

$$\mathrm{IPC}_{\mathsf{Horner}} = \frac{2n}{2n} = 1 \quad \text{instruction par cycle.}$$

Cet IPC idéal de 1 traduit de manière formelle le fait que le schéma de Horner ne présente aucun parallélisme d'instructions.

[1] Cela ne veut pas dire que la valeur de l'IPC soit idéale pour l'utilisateur, bien au contraire.

5.3.4 IPC de CompHorner et de DDHorner

Nous étudions ici l'algorithme CompHorner, lors de l'évaluation d'un polynôme de degré n : dans ce cas, n itérations (numérotées de $n-1$ à 0) de la boucle de CompHorner sont exécutées. Nous cherchons à déterminer la latence totale de l'exécution de cette boucle sur notre processeur idéal, c'est à dire le nombre de cycles nécessaires à l'exécution de n itérations du corps de boucle. Connaissant ce nombre d'instructions flottantes, nous pourrons ensuite en déduire l'IPC idéal de CompHorner.

On considère donc la figure 5.1.a, qui représente le graphe de flot de données pour l'exécution de l'itération i de la boucle de CompHorner. Chaque sommet représenté par un cercle est une opération arithmétique flottante. De plus, il existe un arc de l'opération α vers l'opération β si et seulement si β dépend de α. Ce graphe de flot de données est basé sur le code C listé ci-dessus. Pour aider le lecteur à comprendre comment nous avons

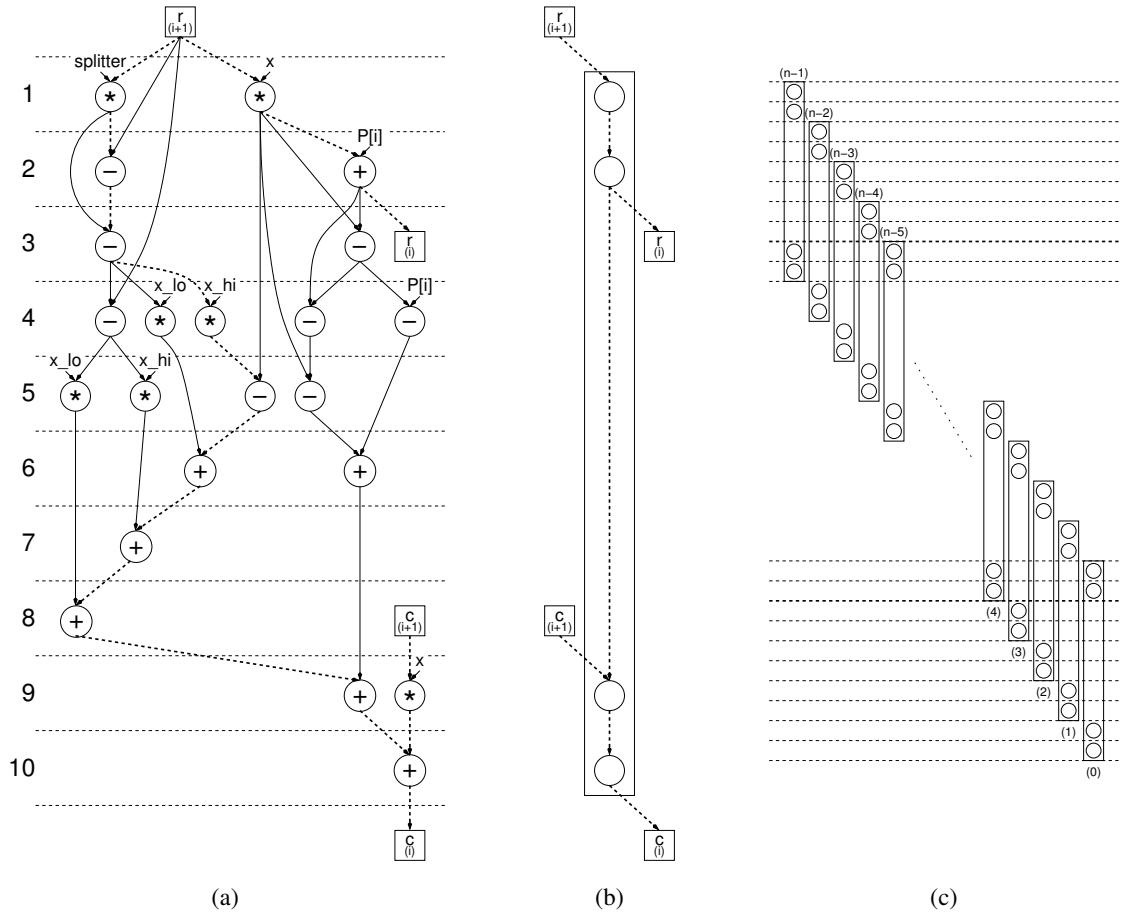


FIG. 5.1 – Graphes de flot de données pour CompHorner.

construit le graphe de la figure 5.1.a, nous reportons dans le tableau 5.3, dans le même ordre que sur la figure, les instructions arithmétiques d'une itération de la boucle. Sur chaque ligne du tableau, nous reportons les instructions pouvant s'exécuter simultanément. Les variables temporaires n'apparaissant pas dans le code sont introduites à l'aide des variables τ indicées.

Dans le graphe de flot de données de la figure 5.1.a, les entrées représentées sous la forme de carrés sont les entrées critiques, puisque ce sont les sorties de l'itération précédente $i+1$.

TAB. 5.3 – Construction du graphe de flot de données pour CompHorner

1	t=r*splitter	p=r*x			
2	τ_1=t-r	r=p+P[i]			
3	hi=t-τ_1	t=r-p			
4	lo=r-hi	τ_2=hi*x_lo	τ_3=hi*x_hi	τ_4=r-t	τ_5=P[i]-t
5	τ_6=lo*xlo	τ_7=lo*x_hi	τ_8=p-τ_3	τ_9=p-τ_4	
6	τ_{10}=τ_2+τ_8	sig=τ_5+τ_9			
7	τ_{11}=τ_7+τ_{10}				
8	pi=τ_{11}+τ_6				
9	τ_{12}=pi+sig	τ_{13}=c*x			
10	c=τ_{12}+τ_{13}				

On doit distinguer principalement trois chemins critiques (représentés en lignes pointillées) dans ce graphe :
- un de r_{i+1} à r_i contenant 2 instructions,
- un de r_{i+1} à c_i contenant 10 instructions,
- un de c_{i+1} à c_i contenant 2 instructions.

Comme le chemin critique de r_{i+1} à c_i contient 10 instructions consécutives, une itération sera exécutée en 10 cycles sur notre processeur idéal. D'après ces observations, on représente sur la figure 5.1.b l'exécution de l'itération i sous la forme d'un rectangle de longueur 10 cycles, et dans lequel :
- r_{i+1} est consommé au premier cycle de l'itération,
- r_i est produit au cycle 2,
- c_{i+1} est consommé au cycle 8,
- et c_i est produit au cycle 10.

Cela nous permet de voir qu'une nouvelle itération peut commencer tous les 2 cycles, et que deux itérations consécutives peuvent se chevaucher de 8 cycles. Sur la figure 5.1.c, on représente ainsi l'exécution de n itérations de la boucle. En s'aidant de ce schéma, on constate facilement que la latence de l'exécution complète de la boucle de CompHorner est $2n + 8$ cycles. Comme l'exécution de cette boucle nécessite $22n$ opérations flottantes, on en déduit que l'IPC idéal de l'algorithme CompHorner est

$$\mathrm{IPC}_{\mathsf{CompHorner}} = \frac{22n}{2n+8} \approx 11 \quad \text{instructions par cycle.}$$

Sur la figure 5.2, on effectue la même analyse pour déterminer la latence de l'exécution complète de la boucle de DDHorner. On représente la graphe de flot de données pour une itération i de la boucle sur la figure 5.2.a. En analysant ce graphe, on représente sur la figure 5.2.b l'exécution de l'itération i comme un rectangle de longueur 19 cycles :
- sh_{i+1} est consommé au premier cycle de l'itération,
- sh_i est produit au cycle 17,
- sl_{i+1} est consommé au cycle 3,
- et sl_i est produit au cycle 19.

On constate donc que l'exécution d'une itération peut commencer tous les 17 cycles, et que deux itérations consécutives se chevauchent de 2 cycles. On représente ainsi sur la

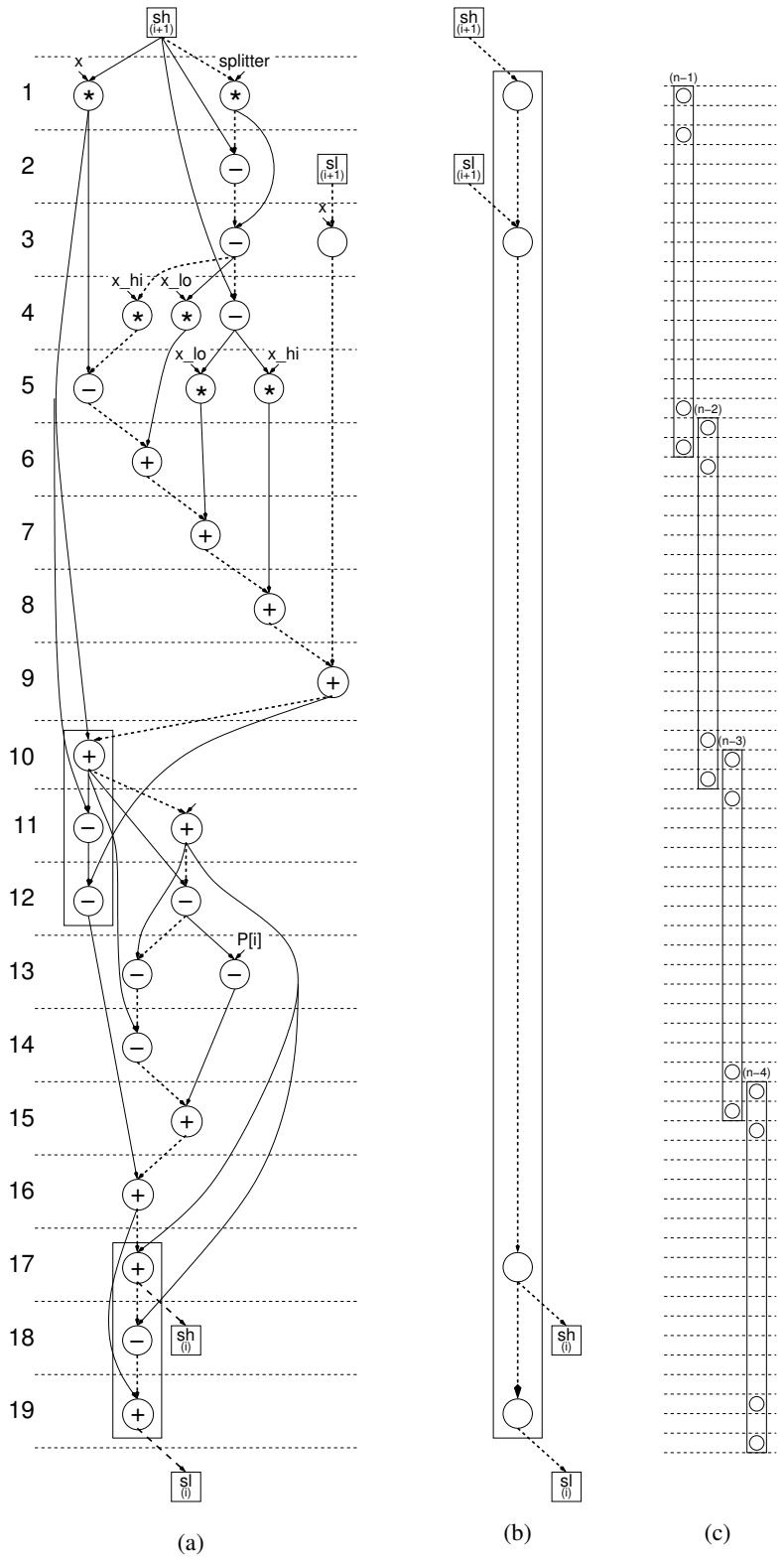


FIG. 5.2 – Graphes de flot de données pour DDHorner.

figure 5.2.c l'exécution des 4 premières itérations. On peut facilement voir que la latence de l'exécution complète de la boucle est de $17n + 2$ cycles d'horloge, Comme on sait par

ailleurs que l'exécution de cette boucle requiert $28n$ opérations flottantes, on en déduit

$$\mathrm{IPC}_{\mathsf{DDHorner}} = \frac{28n}{17n+2} \approx 1.65 \quad \text{instructions par cycle.}$$

5.3.5 Résumé

Nous résumons dans le tableau ci-dessous les différents éléments obtenus au cours de ce chapitre relativement aux performances de nos algorithmes.

	Horner	CompHorner	DDHorner
Nombre d'opérations flottantes	$2n$	$22n+5$	$28n+4$
Surcoût, en terme de nombre d'opérations flottantes, par rapport à Horner	1	≈ 11	≈ 14
Surcoût mesuré en pratique par rapport à Horner	1	$2.7-3.2$	$8.5-9.7$
IPC idéal	1	≈ 11	≈ 1.65

Comme il a déjà été dit, le fait que l'IPC idéal du schéma de Horner soit égal à 1 signifie que cet algorithme ne présente aucun parallélisme d'instruction. On contraire, CompHorner et DDHorner ont tous deux un IPC idéal plus grand que 1 : ces deux algorithmes présentent donc du parallélisme d'instructions, qui sera effectivement exploité par les processeurs superscalaires. Ceci permet d'expliquer le fait que les surcoûts mesurés en pratique pour ces deux algorithmes, comparés au schéma de Horner, sont toujours inférieurs à ceux auxquels on pourrait s'attendre d'après les décomptes d'opérations flottantes.

On constate également que l'IPC idéal de CompHorner est plus de 6 fois supérieur à l'IPC idéal de DDHorner :

$$\mathrm{IPC}_{\mathsf{CompHorner}} \approx 6.66 \times \mathrm{IPC}_{\mathsf{DDHorner}}.$$

La quantité de parallélisme d'instructions exploitable dans une implantation de CompHorner est donc très supérieure à celle disponible dans une implantation de DDHorner. Cela explique, de manière qualitative, les bonnes performances observées pour CompHorner en comparaison de DDHorner.

La conséquence des étapes de renormalisation nécessaires pour les calculs avec les double-doubles apparaît clairement si l'on compare les figures 5.1 et 5.2. Chaque étape de renormalisation est encadrée par un rectangle sur la figure 5.2.a : ces étapes jouent le rôle de « bottlenecks » au cours de l'exécution de DDHorner. En particulier, si l'on compare une itération de la boucle de DDHorner à celle de CompHorner, on constate que :

- la latence de chaque itération de DDHorner est plus longue à cause de la première étape de renormalisation (cycles 10 à 12 sur la figure 5.2.a),
- en raison de la seconde étape de renormalisation (cycles 17 à 19 sur la figure 5.2.a) le chevauchement entre deux itérations consécutives est plus faible.

Le fait que CompHorner évite ces étapes de renormalisation est donc la raison pour laquelle il présente plus de parallélisme d'instructions.

L'étude que nous avons menée ici ne permet pas d'expliquer de façon quantitative le surcoût introduit par chacun des algorithmes CompHorner et DDHorner, pas plus que ne le

permettent les décomptes d'opérations. En effet, l'IPC atteint sur un processeur réaliste, pour une implantation donnée d'un algorithme, sera naturellement toujours inférieur à l'IPC obtenu dans le cadre de notre processeur idéal.

Néanmoins, nous avons mis en évidence le fort potentiel en termes de parallélisme d'instructions de CompHorner par rapport à DDHorner : notre algorithme d'évaluation compensé se montrera donc toujours plus performant que son équivalent en arithmétique double-double sur les processeurs superscalaires à venir et exploitant de plus en plus le parallélisme d'instructions.

5.4 Conclusion

Dans ce chapitre, nous avons montré que le schéma de Horner compensé est une alternative très efficace à l'arithmétique double-double pour doubler la précision de l'évaluation polynomiale. En effet, nos expériences montrent que le schéma de Horner compensé ne s'exécute qu'environ 3 fois plus lentement que le schéma de Horner classique. De plus, le schéma de Horner compensé s'exécute au moins 2 fois plus rapidement que le schéma de Horner en arithmétique double-double, tout en offrant une précision de travail similaire.

Comme nous l'avons souligné dés l'introduction de ce chapitre, les performances de CompHorner face à DDHorner ne peuvent être expliquées uniquement à l'aide de classiques décomptes d'opérations flottantes. Nous avons donc également effectué une analyse détaillée du parallélisme d'instructions disponible dans ces deux algorithmes. Cette étude a montré que le schéma de Horner compensé présente nettement plus de parallélisme d'instructions que son équivalent en arithmétique double-double, ce qui explique de façon qualitative ses bonnes performances en pratique.

En outre, nous avons montré que le défaut de parallélisme d'instructions dans DDHorner est dû aux étapes de renormalisation nécessaires après chaque opération en arithmétique double-double. Autrement dit, l'absence de telles étapes de renormalisation dans CompHorner explique le fait qu'il présente plus de parallélisme d'instructions que DDHorner, et par conséquent son efficacité pratique sur les architectures superscalaires.

La même conclusion s'applique clairement pour les autres algorithmes compensés, qui évitent eux aussi les étapes de renormalisation. Citons en particulier :

- les algorithmes de sommation et de produit scalaire d'Ogita, Rump et Oishi [70],
- les améliorations en présence d'un FMA que nous apportons au schéma de Horner compensé au chapitre suivant,
- le schéma de Horner compensé $K-1$ fois, qui sera présenté au chapitre 8,
- l'algorithme compensé de résolution de systèmes triangulaires, au chapitre 9.

Insistons également sur le fait que l'analyse du parallélisme d'instruction, comme nous l'avons effectuée dans ce chapitre, peut facilement être adaptée, et permettre d'expliquer, ou de comparer, l'efficacité de bien d'autres algorithmes numériques.

Chapitre 6

Amélioration du schéma de Horner compensé avec un FMA

Plan du chapitre : Nous décrivons en Section 6.2 une adaptation de l'algorithme CompHorner (algorithme 4.3 présenté au chapitre 4), utilisant l'algorithme TwoProdFMA pour la compensation des erreurs sur les multiplications. Ensuite, nous décrivons Section 6.3 comment nous avons procédé pour compenser le schéma de Horner avec FMA : cela nous amènera à formuler l'algorithme CompHornerFMA basé sur la transformation exacte ThreeFMA. En Section 6.4, nous comparons les bornes d'erreurs obtenues pour chacun des algorithmes étudiés, et illustrons leur comportement vis à vis du conditionnement par des expériences numériques. Nous comparons finalement les performances de ces algorithmes en Section 6.5.

6.1 FMA et compensation du schéma de Horner

L'instruction « Fused Multiply and Add »(FMA) est disponible sur certains processeurs modernes, comme le Power PC d'IBM ou l'Itanium d'Intel. De plus, sa standardisation est prévue dans le projet [30] de révision de la norme IEEE-754 [41] : il est donc probable que le FMA devienne disponible sur la plupart des processeurs dans un proche avenir.

Étant donnés trois flottants a, b et c, l'instruction FMA calcule $a \times b + c$ avec seulement une erreur d'arrondi finale [61]. Ceci est particulièrement intéressant dans le contexte de l'évaluation de polynômes, dans la mesure où le FMA permet d'exécuter le schéma de Horner plus rapidement.

Supposons que l'on veuille évaluer, en un flottant x donné, un polynôme $p(x) = \sum_{i=0}^{n} a_i x^i$ à coefficient flottants. Tout au long de ce chapitre, l'arrondi au plus proche est utilisé comme mode d'arrondi courant. Si l'on dispose d'un FMA, alors la ligne $\widehat{r}_i = \widehat{r}_{i+1} \otimes x \oplus a_i$ dans le schéma de Horner (algorithme 2.6) peut être remplacée par $\widehat{r}_i = \mathsf{FMA}(\widehat{r}_{i+1}, x, a_i)$. Cela donne le schéma de Horner avec FMA, que nous formulons ci-dessous.

Algorithme 6.1. Schéma de Horner avec FMA.

function $[r_0] = \mathsf{HornerFMA}(p, x)$
  $\widehat{r}_n = a_n$
  for $i = n-1 : -1 : 0$
    $\widehat{r}_i = \mathsf{FMA}(\widehat{r}_{i+1}, x, a_i)$
  end

La borne d'erreur dans le pire cas de l'évaluation polynomiale est améliorée, puisque l'on a maintenant

$$\frac{|p(x) - \mathsf{HornerFMA}(p,x)|}{|p(x)|} \leq \gamma_n \operatorname{cond}(p,x). \tag{6.1}$$

Le facteur apparaissant devant $\operatorname{cond}(p,x)$ est $\gamma_n \approx n\mathbf{u}$ pour HornerFMA, alors que ce même facteur est $\gamma_{2n} \approx 2n\mathbf{u}$ pour Horner. La borne d'erreur dans le pire cas pour le schéma de Horner avec FMA est donc deux fois plus fine que celle du schéma de Horner classique (2.10). Une si faible différence, qui ne correspond qu'à un bit de précision, n'est pas facile à mettre en évidence expérimentalement. Les graphiques où l'on représente la précision relative du résultat calculé en fonction du conditionnement, comme sur la figure 4.1, ne permettent d'observer aucune différence notable entre les comportements numériques des deux algorithmes.

A titre d'exemple, nous reportons sur la figure 6.1 la précision du résultat calculé par les algorithmes Horner et HornerFMA en fonction du nombre de conditionnement. Cette expérience est réalisée en double précision IEEE-754, à l'aide de 300 polynômes de degré 50, produits à l'aide du générateur décrit au chapitre 4. On représente également les bornes d'erreur relative (2.10) et (6.1) pour les deux algorithmes considérés.

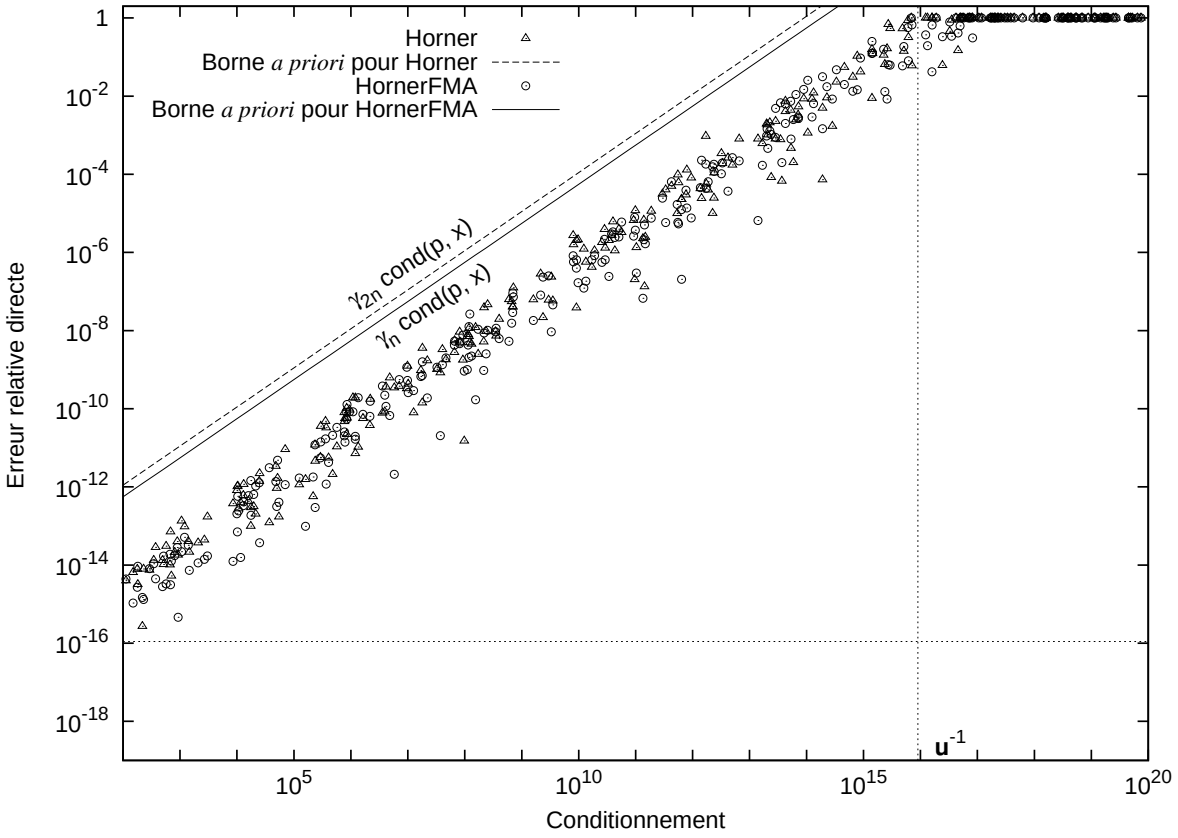


FIG. 6.1 – Précision des résultats calculés par Horner et HornerFMA.

Comme à l'accoutumée, on constate sur la figure 6.1 que les bornes a priori sont toutes deux pessimistes, d'au moins un ordre de grandeur. Il est également important de noter que ce type d'expérience ne permet de mettre en évidence aucune différence de précision en pratique entre les algorithmes Horner et HornerFMA.

Toutefois, le décompte des opérations flottantes nécessitées par l'algorithme 6.1 est deux fois moindre que celui du schéma de Horner classique. Lorsque l'on travaille sur une architecture disposant d'un FMA, on a donc tout intérêt à adapter nos algorithmes de manière à tirer profit de cette instruction. Dans notre contexte, qui est celui des algorithmes compensés, on peut envisager d'utiliser l'instruction FMA d'au moins deux manières.

Premièrement, comme on l'a déjà vu au chapitre 3, le FMA permet d'effectuer la transformation exacte d'une multiplication très efficacement : alors que l'algorithme classique TwoProd (algorithme 3.11) nécessite 17 opérations flottantes, TwoProdFMA (algorithme 3.11) permet le calcul de cette transformation exacte en seulement deux opérations flottantes [46, 61, 68]. On peut donc tirer parti de l'instruction FMA pour compenser plus efficacement les erreurs d'arrondi commises lors des multiplications flottantes, via l'algorithme TwoProdFMA.

D'autre part, une transformation exacte pour le FMA a été proposée par Boldo et Muller [12]. L'algorithme ThreeFMA (algorithme 3.15) a été décrit au chapitre 3. Rappelons brièvement le principe de cette transformation exacte. Étant donnés trois flottants a, b et c, et en supposant que le mode d'arrondi courant est au plus proche, $\mathsf{ThreeFMA}(a, b, c)$ produit trois flottants x, y, et z tels que

$$a \times b + c = x + y + z \quad \text{avec} \quad x = \mathsf{FMA}(a, b, c).$$

On pourra utiliser la transformation exacte ThreeFMA pour compenser les erreurs d'arrondi entachant les opérations FMA.

Dans ce chapitre, nous envisageons ces deux alternatives, dans le cas de la compensation du schéma de Horner. Nous décrivons :

- l'algorithme $\mathsf{CompHorner}_{\mathsf{fma}}$, dont le principe est identique à celui de CompHorner, mais dans lequel les erreurs d'arrondi sur les multiplications sont compensées à l'aide de TwoProdFMA,
- l'algorithme CompHornerFMA, qui est une version compensée de HornerFMA, les erreurs d'arrondi entachant les opérations FMA étant calculées grâce à ThreeFMA.

Dans les deux cas, nous montrons que l'utilisation du FMA n'améliore pas de façon significative la précision du résultat compensée. Nous verrons néanmoins que l'approche basée sur la compensation des erreurs d'arrondi à l'aide de TwoProdFMA est en pratique l'alternative la plus performante pour doubler la précision des calculs en présence d'un FMA.

6.2 Utiliser le FMA pour compenser les multiplications

Dans le cas du schéma de Horner, utiliser le FMA pour compenser les multiplications revient à remplacer, dans l'algorithme EFTHorner (algorithme 4.1), la transformation exacte TwoProd par TwoProdFMA. On se contente donc ci-dessous de reformuler EFTHorner, en lui attribuant le nom $\mathsf{EFTHorner}_{\mathsf{fma}}$: en l'absence d'underflow, l'algorithme 6.2 produit les mêmes résultats que l'algorithme 4.1. Néanmoins, comme on utilise ici TwoProdFMA au lieu de TwoProd, l'algorithme $\mathsf{EFTHorner}_{\mathsf{fma}}$ ne nécessite plus que $8n$ opérations flottantes.

Algorithme 6.2. Transformation exacte pour le schéma de Horner utilisant TwoProdFMA

function $[\widehat{r}_0, p_\pi, p_\sigma] = \mathsf{EFTHorner}_{\mathsf{fma}}(p, x)$
  $\widehat{r}_n = a_n$
  for $i = n - 1 : -1 : 0$

$[p_i, \pi_i] = \mathsf{TwoProdFMA}(\widehat{r}_{i+1}, x)$
$[\widehat{r}_i, \sigma_i] = \mathsf{TwoSum}(p_i, a_i)$
Soit π_i le coefficient de degré i du polynôme p_π
Soit σ_i le coefficient de degré i du polynôme p_σ
end

Puisque nous disposons d'un FMA, nous adaptons également la manière dont est évalué le terme correctif $\widehat{c}$. Dans l'algorithme 4.3, on remplace donc la ligne $\widehat{c} = \mathsf{Horner}(p_\pi \oplus p_\sigma, x)$ (algorithme 2.6) par $\widehat{c} = \mathsf{HornerFMA}(p_\pi \oplus p_\sigma, x)$ (algorithme 6.1). Cette adaptation du schéma de Horner compensé, formulée ci-dessous, ne nécessite plus que $10n + \mathcal{O}(1)$ opérations flottantes pour s'exécuter.

Algorithme 6.3. Schéma de Horner compensé en présence d'un FMA.

function $\overline{r} = \mathsf{CompHorner}_{\mathsf{fma}}(p, x)$
$[\widehat{r}, p_\pi, p_\sigma] = \mathsf{EFTHorner}_{\mathsf{fma}}(p, x)$
$\widehat{c} = \mathsf{HornerFMA}(p_\pi \oplus p_\sigma, x)$
$\overline{r} = \widehat{r} \oplus \widehat{c}$

Il est évident que l'algorithme 6.3 est au moins aussi précis que l'algorithme CompHorner : l'évaluation produite par cet algorithme satisfait toujours la borne d'erreur du théorème 4.4. Le théorème 6.4 nous fournit une borne d'erreur *a priori* plus fine pour $\mathsf{CompHorner}_{\mathsf{fma}}$. Nous insisterons sur ce point en Section 6.4.

Théorème 6.4. *On considère un polynôme p à coefficients flottants de degré n et x un nombre flottant. Alors, en l'absence d'underflow, on a*

$$|\mathsf{CompHorner}_{\mathsf{fma}}(p, x) - p(x)| \leq \mathbf{u}|p(x)| + (1 + \mathbf{u})\gamma_n\gamma_{2n}\widetilde{p}(x). \tag{6.2}$$

Pour la preuve du théorème 6.4, nous commençons par formuler le lemme suivant, qui est l'analogue du lemme 4.5 en présence d'un FMA.

Lemme 6.5. *Soient p et q deux polynômes à coefficients flottants, de degré n, tels que $p(x) = \sum_{i=0}^{n} a_i x^i$ et $q(x) = \sum_{i=0}^{n} b_i x^i$. On considère l'évaluation flottante de $(p + q)(x)$ calculée par $\mathsf{HornerFMA}(p \oplus q, x)$. Alors, en l'absence d'underflow,*

$$|(p + q)(x) - \mathsf{HornerFMA}(p \oplus q, x)| \leq \gamma_{n+1}(\widetilde{p + q})(x). \tag{6.3}$$

Le lemme 6.5 permet de déduire facilement une borne sur l'erreur $|\widehat{c} - c|$ entachant le terme correctif calculé. Nous pouvons maintenant formuler la preuve du théorème 6.4, similaire à celle du théorème 4.4.

Preuve du théorème 6.4. Comme dans la preuve du théorème 4.4, on a

$$|\overline{r} - p(x)| \leq \mathbf{u}|p(x)| + (1 + \mathbf{u})|\widehat{c} - c|.$$

Comme $\widehat{c} = \mathsf{HornerFMA}(p_\pi \oplus p_\sigma, x)$ avec p_π et p_σ deux polynômes de degré $n - 1$, le lemme 6.5 donne $|\widehat{c} - c| \leq \gamma_n(\widetilde{p_\pi + p_\sigma})(x)$. Rappelons ici que, d'après l'inégalité (4.6), on a $(\widetilde{p_\pi + p_\sigma})(x) \leq \gamma_{2n}\widetilde{p}(x)$, donc $|\widehat{c} - c| \leq \gamma_n\gamma_{2n}\widetilde{p}(x)$. On obtient ainsi la majoration (6.2). ■

6.3 Compensation du schéma de Horner avec FMA

Le principe de l'algorithme CompHornerFMA que nous formulons dans cette section est le même que celui du schéma de Horner décrit au chapitre 4. La différence tient au fait qu'au lieu de compenser les erreurs d'arrondi commises lors de l'évaluation du polynôme par l'algorithme Horner (algorithme 2.6), nous allons compenser celles commises par HornerFMA (algorithme 6.1).

6.3.1 Une transformation exacte pour HornerFMA

Soit $p(x) = \sum_{i=0}^{n} a_i x^i$ un polynôme de degré n à coefficients flottants, et soit x un nombre flottant. Nous proposons ci-dessous une transformation exacte pour l'évaluation de $p(x)$ par le schéma de Horner avec FMA. Cet algorithme requiert $17n$ opérations flottantes.

Algorithme 6.6. Transformation exacte pour le schéma de Horner avec FMA

function $[\widehat{r}_0, p_\varepsilon, p_\varphi] = \mathsf{EFTHornerFMA}(p, x)$
  $\widehat{r}_n = a_n$
  for $i = n-1 : -1 : 0$
    $[\widehat{r}_i, \varepsilon_i, \varphi_i] = \mathsf{ThreeFMA}(\widehat{r}_{i+1}, x, a_i)$
    Soit ε_i le coefficient de degré i du polynôme p_ε
    Soit φ_i le coefficient de degré i du polynôme p_φ
  end

Théorème 6.7. *Soit $p(x) = \sum_{i=0}^{n} a_i x^i$ un polynôme de degré n à coefficients flottants, et soit x un nombre flottant. En l'absence d'underflow, l'algorithme 6.6 calcule à la fois $\widehat{r}_0 = \mathsf{HornerFMA}(p, x)$ et deux polynômes p_ε et p_φ de degré $n-1$ à coefficients flottants, tels que*

$$p(x) = \mathsf{HornerFMA}(p, x) + (p_\varepsilon + p_\varphi)(x). \tag{6.4}$$

De plus,

$$\widetilde{(p_\varepsilon + p_\varphi)}(x) \leq \gamma_n \widetilde{p}(x). \tag{6.5}$$

À nouveau, insistons sur le fait que la relation (6.4) signifie que l'algorithme 6.6 est une transformation exacte pour l'évaluation polynomiale avec le schéma de Horner.

Preuve du théorème 6.7. Comme ThreeFMA est une transformation exacte, d'après le théorème 3.16, on a $\widehat{r}_i = \widehat{r}_{i+1} x + a_i - \varepsilon_i - \varphi_i$, pour $i = 0, \ldots, n-1$. Comme $\widehat{r}_n = a_n$, on peut montrer par récurrence que

$$\widehat{r}_0 = \sum_{i=0}^{n} a_i x^i - \sum_{i=0}^{n-1} (\varepsilon_i + \varphi_i) x^i,$$

ce qui démontre la relation (6.4).

Démontrons maintenant la relation (6.5). Comme $[\widehat{r}_i, \varepsilon_i, \varphi_i] = \mathsf{ThreeFMA}(\widehat{r}_{i+1}, x, a_i)$, d'après le théorème 3.16, on a $|\varepsilon_i + \varphi_i| \leq \mathbf{u}|\widehat{r}_i|$. On en déduit que

$$\widetilde{(p_\varepsilon + p_\varphi)}(x) = \sum_{i=0}^{n-1} |\varepsilon_i + \varphi_i||x^i| = \sum_{i=1}^{n} |\varepsilon_{n-i} + \varphi_{n-i}||x^{n-i}| \leq \mathbf{u} \sum_{i=1}^{n} |\widehat{r}_{n-i}||x^{n-i}|. \tag{6.6}$$

Pour $i = 1, \ldots, n$ on a $\widehat{r}_{n-i} = \mathsf{FMA}(\widehat{r}_{n-i+1}, x, a_{n-i}) = \langle 1 \rangle (\widehat{r}_{n-i+1} x + a_{n-i})$. Comme de plus $\widehat{r}_n = a_n$, pour $i = 1, \ldots, n$ on a

$$\widehat{r}_{n-i} = \langle i \rangle a_n x^i + \langle i \rangle a_{n-1} x^{i-1} + \langle i-1 \rangle a_{n-2} x^{i-2} + \cdots + \langle 2 \rangle a_{n-i+1} x + \langle 1 \rangle a_{n-i}.$$

Puisque chacun des compteurs d'erreur $\langle k \rangle$ est un facteur de la forme $(1+\theta_k)$, avec $|\theta_k| \leq \gamma_k$, on a

$$|\widehat{r}_{n-i}| \leq (1+\gamma_i) \sum_{j=0}^{i} |a_{n-i+j}||x^j|.$$

D'après (6.6), on obtient donc

$$\widetilde{(p_\varepsilon + p_\varphi)}(x) \leq \mathbf{u}(1+\gamma_n) \sum_{i=1}^{n} \sum_{j=0}^{i} |a_{n-i+j}||x^{n-i+j}| \leq n\mathbf{u}(1+\gamma_n)\widetilde{p}(x).$$

En remarquant que $n\mathbf{u}(1+\gamma_n) = \gamma_n$, on obtient le résultat annoncé. ■

6.3.2 L'algorithme CompHornerFMA

Puisque nous disposons de la transformation exacte EFTHornerFMA, nous raisonnons comme dans le cas du schéma de Horner compensé. L'erreur directe entachant l'évaluation de $p(x)$ avec HornerFMA est exactement

$$c = p(x) - \mathsf{HornerFMA}(p, x) = (p_\varepsilon + p_\varphi)(x),$$

où les deux polynômes p_ε et p_φ sont calculés exactement à l'aide de l'algorithme 6.6. On calcule donc une approximation $\widehat{c}$ de cette erreur directe c, afin de produire un résultat compensé $\overline{r}$, tel que $\overline{r} = \mathsf{HornerFMA}(p, x) \oplus \widehat{c}$. Nous utilisons bien sûr le schéma de Horner avec FMA pour calculer le terme correctif approché $\widehat{c}$.

Algorithme 6.8. Schéma de Horner avec FMA compensé.

function $\overline{r} = \mathsf{CompHornerFMA}(p, x)$
  $[\widehat{r}, p_\varepsilon, p_\varphi] = \mathsf{EFTHornerFMA}(p, x)$
  $\widehat{c} = \mathsf{HornerFMA}(p_\varepsilon \oplus p_\varphi, x)$
  $\overline{r} = \widehat{r} \oplus \widehat{c}$

L'algorithme CompHornerFMA requiert $19n$ opérations flottantes. Le théorème suivant fournit une borne sur l'erreur entachant le résultat compensé calculé par cette méthode.

Théorème 6.9. *On considère un polynôme p à coefficients flottants de degré n et x un nombre flottant. Alors, en l'absence d'underflow on a*

$$|\mathsf{CompHornerFMA}(p, x) - p(x)| \leq \mathbf{u}|p(x)| + (1+\mathbf{u})\gamma_n^2 \widetilde{p}(x). \tag{6.7}$$

Preuve. Puisque la transformation EFTHornerFMA est exacte, on a

$$|\overline{r} - p(x)| \leq \mathbf{u}|p(x)| + (1+\mathbf{u})|\widehat{c} - c|.$$

Puisque $\widehat{c} = \mathsf{HornerFMA}(p_\varepsilon \oplus p_\varphi, x)$, comme p_ε et p_φ sont deux polynômes de degré $n-1$, en utilisant le lemme 6.5 on obtient

$$|\overline{r} - p(x)| \leq \mathbf{u}|p(x)| + (1+\mathbf{u})\gamma_n \widetilde{(p_\varepsilon + p_\varphi)}(x).$$

D'après le théorème 6.7, $\widetilde{(p_\varepsilon + p_\varphi)}(x) \leq \gamma_n \widetilde{p}(x)$, ce qui permet d'obtenir (6.7). ■

6.4 Synthèse des résultats

Nous avons décrit jusqu'ici trois algorithmes compensés pour effectuer une évaluation polynomiale en précision doublée :

- au chapitre 4, l'algorithme CompHorner (algorithme 4.3),
- les algorithmes $\mathsf{CompHorner}_{\mathsf{fma}}$ (algorithme 6.3) et CompHornerFMA (algorithme 6.8), conçus pour tirer parti de l'instruction FMA.

Dans cette section, nous regroupons les bornes d'erreur obtenues, et tentons de comparer la précision effective de chacun de ces algorithmes. Nous regroupons également les différents décomptes d'opérations.

6.4.1 Borne d'erreur et précision du résultat calculé

Pour interpréter les bornes d'erreur *a priori* obtenues pour les algorithmes CompHorner, $\mathsf{CompHorner}_{\mathsf{fma}}$ et CompHornerFMA, nous faisons intervenir le nombre de conditionnement $\mathrm{cond}(p,x)$. Cela nous donne une borne sur l'erreur relative $|\overline{r}-p(x)|/|p(x)|$ entachant le résultat compensé $\overline{r}$ calculé par chacun de ces algorithmes. Ces bornes d'erreur relative sont regroupées dans le tableau 6.1.

TAB. 6.1 – Bornes d'erreur relatives obtenues pour CompHorner, $\mathsf{CompHorner}_{\mathsf{fma}}$ et CompHornerFMA.

Algorithme		Borne d'erreur relative *a priori*
CompHorner	(algorithme 4.3)	$\mathbf{u}+\gamma_{2n}^2\,\mathrm{cond}(p,x)$ $\approx \mathbf{u}+4n^2\mathbf{u}^2\,\mathrm{cond}(p,x)$
$\mathsf{CompHorner}_{\mathsf{fma}}$	(algorithme 6.3)	$\mathbf{u}+(1+\mathbf{u})\gamma_{2n}\gamma_n\,\mathrm{cond}(p,x)$ $\approx \mathbf{u}+2n^2\mathbf{u}^2\,\mathrm{cond}(p,x)$
CompHornerFMA	(algorithme 6.8)	$\mathbf{u}+\gamma_n^2\,\mathrm{cond}(p,x)$ $\approx \mathbf{u}+n^2\mathbf{u}^2\,\mathrm{cond}(p,x)$

On constate que les bornes reportées dans le tableau 6.1 sont toutes de la forme

$$\frac{|\overline{r}-p(x)|}{|p(x)|} \leq \mathbf{u}+\mathcal{O}(u^2)\,\mathrm{cond}(p,x).$$

On en déduit que lorsque $\mathrm{cond}(p,x) \ll \mathbf{u}^{-1}$, le premier terme $\mathbf{u}$ est prédominant devant $\mathcal{O}(u^2)\,\mathrm{cond}(p,x)$. Dans ce cas, les trois bornes sont donc quasiment égales, et sont toutes de l'ordre de grandeur de l'unité d'arrondi $\mathbf{u}$.

Par contre, lorsque $\mathbf{u}^{-1} \ll \mathrm{cond}(p,x) \lesssim \mathbf{u}^{-2}$, le second terme $\mathcal{O}(u^2)\,\mathrm{cond}(p,x)$ est prépondérant devant $\mathbf{u}$, et les trois bornes d'erreurs se différencient. Les ordres de grandeurs reportés dans le tableau 6.1 nous montrent qu'alors :

1. la borne d'erreur *a priori* obtenue pour $\mathsf{CompHorner}_{\mathsf{fma}}$ est environ deux fois plus fine que celle pour CompHorner.
2. la borne d'erreur pour CompHornerFMA est elle aussi deux fois plus fine que celle obtenue pour $\mathsf{CompHorner}_{\mathsf{fma}}$.

En comparant les preuves des théorèmes 4.4, 6.4 et 6.9, on constate que la première amélioration d'un facteur deux est due à l'utilisation de HornerFMA au lieu de Horner pour l'évaluation du terme correctif $(p_\pi + p_\sigma)(x)$. La seconde amélioration d'un facteur deux tient à l'utilisation de HornerFMA à la fois pour l'évaluation du polynôme initial $p(x)$ et du terme correctif. Lorsque $\mathbf{u}^{-1} \ll \mathrm{cond}(p, x) \lesssim \mathbf{u}^{-2}$, on peut donc s'attendre à ce que l'algorithme CompHornerFMA produise un résultat généralement plus précis que $\mathsf{CompHorner}_{\mathsf{fma}}$, et à ce que $\mathsf{CompHorner}_{\mathsf{fma}}$ soit à son tour plus précis que CompHorner.

Pour éclairer notre propos, nous effectuons la même expérience que celle rapportée en introduction pour les algorithmes Horner et CompHorner. Nous reportons sur la figure 6.2 la précision des évaluations calculées par les algorithmes CompHorner, $\mathsf{CompHorner}_{\mathsf{fma}}$ et HornerFMA en fonction du nombre de conditionnement. Cette expérience est réalisée en double précision IEEE-754, sur un échantillon de 300 polynômes de degré 50 produits à l'aide du générateur du chapitre 4. On représente également les bornes d'erreur relatives du tableau 6.1, pour chacun des algorithmes considérés.

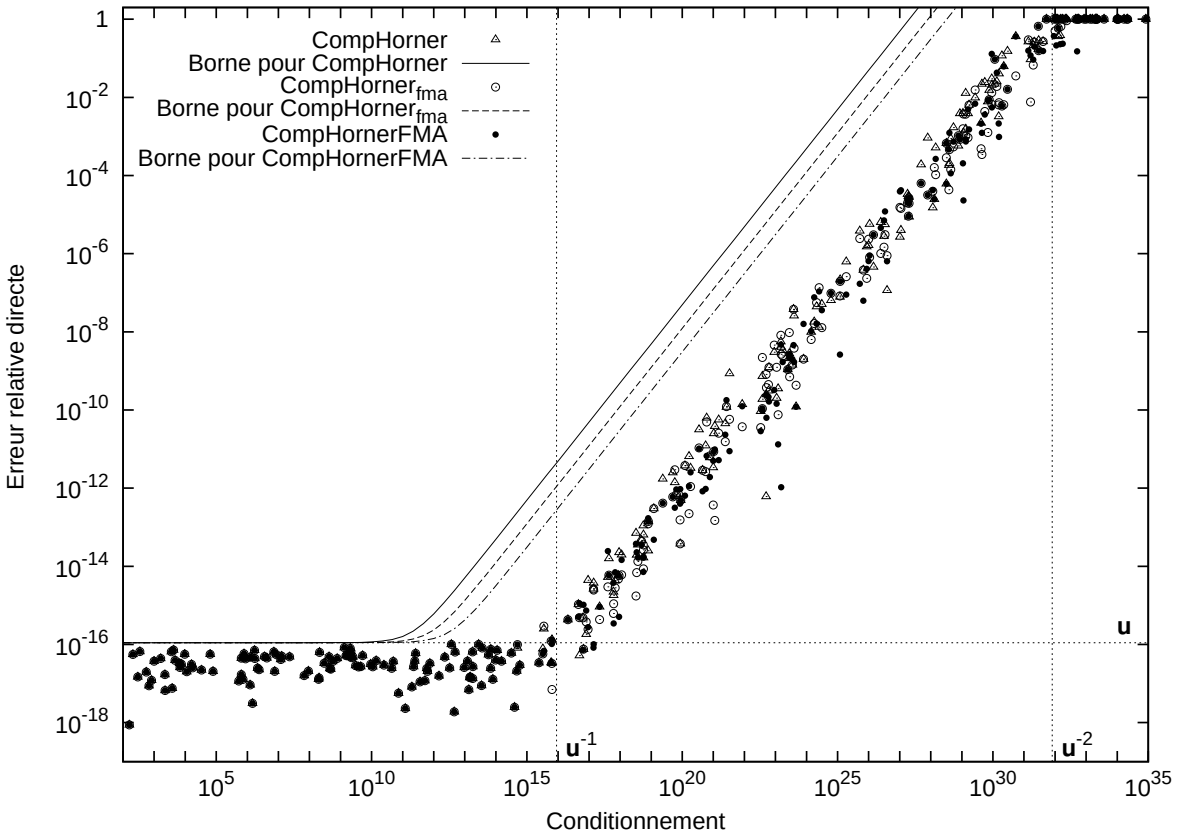


FIG. 6.2 – Précision des résultats calculés par Horner et HornerFMA.

À nouveau, on constate sur la figure 6.2 que les bornes a priori sont pessimistes de plusieurs ordres de grandeurs, et que les trois algorithmes considérés ne peuvent être différenciés du point de vue de la précision du résultat calculé, tout du moins dans ces expériences.

6.4.2 Décompte des opérations flottantes

Pour mémoire, nous regroupons dans le tableau 6.2 les décomptes des opérations flottantes, pour les transformations exactes pour l'évaluation par le schéma de Horner. Ce tableau regroupe les résultats obtenus au chapitre 4 et dans le présent chapitre.

TAB. 6.2 – Décompte des opérations flottantes pour les transformations exactes étudiées.

Algorithme		Nombre d'opérations flottantes
EFTHorner	(algorithme 4.1)	$19n + 4$
$\mathsf{EFTHorner}_{\mathsf{fma}}$	(algorithme 6.2)	$8n$
EFTHornerFMA	(algorithme 6.6)	$17n$

Dans le tableau 6.3, nous regroupons également les décomptes d'opérations flottantes pour chacun des algorithmes présentés jusqu'ici pour d'effectuer une évaluation polynomiale en précision doublée.

TAB. 6.3 – Décompte des opérations flottantes pour les différents algorithmes d'évaluation polynomiale en précision doublée.

Algorithme		Nombre d'opérations flottantes
CompHorner	(algorithme 4.3)	$22n + \mathcal{O}(1)$
DDHorner	(algorithme 3.27)	$29n + \mathcal{O}(1)$
$\mathsf{CompHorner}_{\mathsf{fma}}$	(algorithme 6.3)	$10n + \mathcal{O}(1)$
CompHornerFMA	(algorithme 6.8)	$19n + \mathcal{O}(1)$
DDHorner avec FMA	(algorithme 3.27)	$16n + \mathcal{O}(1)$

6.5 Performances en pratique

Nous réalisons nos mesures sur l'architecture Itanium, c'est à dire dans les environnements (VI) et (VI) listés dans le tableau 5.1. Les mesures sont effectuées de la même manière que celles réalisées au chapitre 5. Nous utilisons un jeu de 39 polynômes, dont le degré varie de 10 à 200 par pas de 5, et pour chaque degré on mesure le ratio du temps d'exécution de CompHorner sur le temps d'exécution du schéma de Horner avec FMA. On procède de même pour les algorithmes CompHornerFMA et DDHorner. Rappelons qu'en présence d'un FMA, nous utilisons toujours une implantation de l'algorithme DDHorner dans laquelle TwoProd (algorithme 3.9) est remplacé par TwoProdFMA (algorithme 3.11) — voir chapitre 3, Section 3.4. Nous reportons dans le tableau 6.4 la moyenne des ratios mesurés.

Observons tous d'abord que nos deux algorithmes compensés, $\mathsf{CompHorner}_{\mathsf{fma}}$ et CompHornerFMA, s'exécutent tous deux significativement plus rapidement que DDHorner.

Comme à l'accoutumée, on constate que les surcoûts mesurés sont toujours plus faibles que ceux auxquels on aurait pu s'attendre d'après le décompte des opérations flottantes. En effet, si l'on ne considère que ces décomptes, d'après le tableau 6.3 on a

$$\frac{\mathsf{CompHorner}_{\mathsf{fma}}}{\mathsf{HornerFMA}} \approx 10, \quad \frac{\mathsf{CompHornerFMA}}{\mathsf{HornerFMA}} \approx 19, \quad \text{et} \quad \frac{\mathsf{DDHorner}}{\mathsf{HornerFMA}} \approx 16.$$

TAB. 6.4 – Performances de CompHorner, CompHornerFMA et DDHorner.

environnement	$\frac{\text{CompHorner}_{\text{fma}}}{\text{HornerFMA}}$	$\frac{\text{CompHornerFMA}}{\text{HornerFMA}}$	$\frac{\text{DDHorner}}{\text{HornerFMA}}$
(VI) Itanium, gcc	2.8	5.3	6.7
(VII) Itanium, icc	1.5	1.8	5.9

Il est intéressant de remarquer que, d'après ces décomptes d'opérations flottantes, on pourrait s'attendre à ce que DDHorner s'exécute plus rapidement que CompHornerFMA, alors que l'on observe l'inverse : CompHornerFMA s'exécute en pratique plus rapidement que DDHorner, alors que DDHorner nécessite moins d'opérations flottantes. Cela souligne le fait que les résultats du chapitre 5 se généralisent en présence d'un FMA : CompHornerFMA présente plus de parallélisme d'instructions que DDHorner, et s'exécute en pratique plus rapidement sur les architectures conçues pour exploiter ce parallélisme d'instructions.

D'autre part, $\text{CompHorner}_{\text{fma}}$ est en pratique l'alternative la plus efficace pour doubler la précision de l'évaluation polynomiale : $\text{CompHorner}_{\text{fma}}$ s'exécute plus de deux fois plus rapidement que DDHorner. Retenons donc qu'en présence d'un FMA, il s'avère plus intéressant d'utiliser la version compensée de Horner que de compenser HornerFMA.

6.6 Conclusion

Nous avons présenté deux versions adaptées du schéma de Horner compensé en présence d'un FMA. Résumons de ces deux algorithmes.

- Dans $\text{CompHorner}_{\text{fma}}$, nous avons compensé, à l'aide de TwoProdFMA, les erreurs d'arrondi entachant chaque multiplication effectuée par Horner.
- Dans CompHornerFMA, nous avons utilisé l'algorithme ThreeFMA pour compenser les opérations FMA effectuées par HornerFMA.

Pour ces deux algorithmes, nous avons obtenu respectivement deux nouvelles bornes d'erreur, plus fines que celle obtenue pour l'algorithme initial. Ces améliorations de la borne d'erreur dans le pire cas s'avèrent toutefois trop faibles pour être observées dans nos expériences numériques — nous avons vu que c'était aussi le cas pour les schémas non compensés. Nous retiendrons donc simplement que les algorithmes $\text{CompHorner}_{\text{fma}}$ et CompHornerFMA présentent la même précision que l'algorithme CompHorner : le résultat est aussi précis que s'il avait été calculé en précision doublée, avec un arrondi final vers le précision de travail.

Il est à noter que ces deux améliorations du schéma de Horner compensé s'exécutent toutes deux significativement plus rapidement que le schéma de Horner basé sur l'arithmétique double-double, qui pourtant tire également parti de la présence d'un FMA. Cela s'explique clairement d'après l'étude du parallélisme d'instructions menée au chapitre 5.

En tout état de cause, l'algorithme $\text{CompHorner}_{\text{fma}}$ est en pratique l'alternative la plus efficace pour simuler une précision de travail doublée en présence d'un FMA. En effet, ce schéma d'évaluation compensé n'introduit qu'un surcoût très modéré par rapport à l'algorithme de Horner avec FMA, et s'exécute au moins 2 fois plus rapidement que le schéma de Horner en arithmétique double-double.

En conclusion, l'étude présentée dans ce chapitre montre qu'il est préférable de tirer parti de la présence du FMA au travers de la transformation exacte TwoProdFMA, afin de compenser les erreurs d'arrondi générées par les multiplications, plutôt que de compenser les erreurs d'arrondi générées par le FMA à l'aide de ThreeFMA.

Chapitre 7

Validation de l'évaluation polynomiale compensée

Plan du chapitre : Dans la Section 7.2, nous montrons comment valider le résultat de l'évaluation compensée de $p(x)$. La Section 7.3 est consacrée aux expériences numériques, afin d'illustrer la finesse de la borne *a posteriori* obtenue, ainsi que le comportement du test dynamique de l'arrondi fidèle. Les tests de la Section 7.4 montrent les bonnes performances pratiques de la version validée du schéma de Horner compensé. Enfin nous traitons en Section 7.5 le cas de l'underflow.

7.1 Qu'entend-on par « validation » ?

Au chapitre 4, nous avons introduit le schéma de Horner compensé. La borne d'erreur *a priori* (4.9) sur l'erreur entachant le résultat calculé par CompHorner nous a permis de comprendre le comportement numérique de cet algorithme : le schéma de Horner compensé est aussi précis que le schéma de Horner classique utilisé en précision doublée, avec un arrondi final vers la précision courante. Le théorème 4.10 fournit également une condition suffisante pour garantir le fait que $\overline{r}$ soit un arrondi fidèle de $p(x)$.

Nous avons déjà souligné, dans les expériences numériques de la Section 4.5, le caractère pessimiste de la borne *a priori* (4.9) : l'erreur directe entachant le résultat compensé est très largement surestimée par cette borne. Citons à ce sujet Higham [39, p.65] :

> The constant terms in an error bound (those depending only on the problem dimensions) are the least important part of it. [...], the constants usually cause the bound to overestimate the actual error by orders of magnitude. [...] If sharp error estimates or bounds are desired they should be computed a posteriori, so that the actual rounding errors that occur can be taken into account.

De plus, la borne *a priori* (4.9) fait intervenir le résultat exact $p(x)$, qui est par définition inconnu, et ne peut donc pas être évaluée.

Dans ce chapitre, nous montrons comment calculer une borne d'erreur *a posteriori* sur le résultat calculé par le schéma de Horner compensé, en supposant dans un premier temps qu'aucun underflow n'intervient au cours des calculs. Comme dans la citation de Higham ci-dessus, l'adjectif *a posteriori* est utilisé pour indiquer que

– le calcul de cette borne vient s'ajouter au calcul du résultat compensé,
– la borne d'erreur n'est connue qu'après l'exécution de l'algorithme.

Plus formellement, étant donné p un polynôme à coefficients flottants, x un flottant et $\overline{r} = \mathsf{CompHorner}(p, x)$, nous proposons un algorithme pour calculer à la fois $\overline{r}$ et $\beta \in \mathbf{F}$ tel que

$$|\overline{r} - p(x)| \leq \beta. \tag{7.1}$$

L'algorithme que nous proposons ne repose que sur des opérations élémentaires en arithmétique flottante, dans le mode d'arrondi au plus proche : nous n'utilisons ni bibliothèque pour le calcul d'intervalles, ni changement du mode d'arrondi.

Il est à noter que l'évaluation de cette borne d'erreur *a posteriori* est elle-même sujette aux erreurs d'arrondi, comme la plupart des calculs effectués en arithmétique flottante. Nous prendrons donc en compte les erreurs d'arrondi commises lors du calcul de β, de manière prouver rigoureusement l'inégalité (7.1). Insistons donc sur le fait que β ne constitue pas une simple estimation, mais bien une borne sur l'erreur directe $|\overline{r} - p(x)|$.

Nous montrerons également comment tester *a posteriori* si l'évaluation compensée est fidèle. Nous formulerons l'algorithme CompHornerBound (algorithme 7.3), qui, en plus du résultat compensé $\overline{r} = \mathsf{CompHorner}(p, x)$, calcule une borne d'erreur *a posteriori* pour $\overline{r}$, et teste dynamiquement si $\overline{r}$ est un arrondi fidèle de $p(x)$.

Il est donc possible de s'assurer de la qualité de l'évaluation compensée $\overline{r}$ produite par l'algorithme CompHornerBound à l'aide de la borne d'erreur β, ou du test dynamique de l'arrondi fidèle : CompHornerBound est donc un algorithme validé [83, 84].

Pour tous les résultats énoncés jusqu'ici, nous avons supposé qu'aucun dépassement de capacité n'intervenait au cours des calculs. Comme dans le chapitre 4, nous proposerons aussi un algorithme pour le calcul d'une borne d'erreur *a posteriori* prenant en compte la possibilité d'underflow.

7.2 Validation de l'évaluation compensée

Nous proposons ici une version validée du schéma de Horner compensé. Tout au long de cette section, nous utilisons les mêmes notations que dans le chapitre 4, que nous rappelons néanmoins brièvement ci-après. On considère donc un polynôme p de degré n à coefficients flottants, et x un argument flottant.

– On note $[\widehat{r}, p_\pi, p_\sigma] = \mathsf{EFTHorner}(p, x)$ (algorithme 4.1), où $\widehat{r} = \mathsf{Horner}(p, x)$ (algorithme 2.6), et p_π et p_σ sont deux polynômes de degré $n-1$ à coefficients flottants.
– $c = (p_\pi + p_\sigma)(x)$ désigne le terme correctif exact pour $\widehat{r}$.
– $\widehat{c} = \mathsf{Horner}(p_\pi \oplus p_\sigma, x)$ est le terme correctif calculé.
– $\overline{r} = \mathsf{CompHorner}(p, x)$ (algorithme 4.3) est le résultat compensé, tel que $\overline{r} = \widehat{r} \oplus \widehat{c}$.

Les résultats de cette section sont tous obtenus en supposant qu'aucun underflow n'intervient au cours des calculs flottants.

7.2.1 Borne d'erreur *a posteriori* pour le terme correctif

Le lemme suivant fournit une borne *a posteriori* sur l'erreur entachant le terme correctif calculé $\widehat{c}$. Insistons sur le fait que cette borne est bien un majorant de l'erreur $|\widehat{c} - c|$, même lorsqu'elle est évaluée en arithmétique flottante.

Lemme 7.1. *On considère un polynôme p de degré n à coefficients flottants et x un flottant. On utilise les notations définies ci-dessus, et on suppose $2(n+1)\mathbf{u} < 1$. L'erreur entachant le terme correctif calculé est majorée comme suit,*

$$|\widehat{c} - c| \leq (\widehat{\gamma}_{2n-1} \otimes \mathsf{Horner}(|p_\pi \oplus p_\sigma|, |x|)) \oslash (1 - 2(n+1)\mathbf{u}) =: \alpha. \tag{7.2}$$

Le lemme 7.1 nous fournit une borne validée α sur l'erreur $|\widehat{c} - c|$ entachant le terme correctif calculé. D'après le lemme 4.9, on en déduit facilement une condition suffisante pour tester si $\widehat{r}$ est un arrondi fidèle : si $\alpha < \frac{\mathbf{u}}{2}|\overline{r}|$, alors $\overline{r}$ est un arrondi fidèle de $p(x)$.

Ce test dynamique est bien entendu incomplet : s'il est satisfait, cela prouve que $\overline{r}$ est un arrondi fidèle de $p(x)$. Dans le cas contraire, on ne peut rien conclure : $\overline{r}$ peut, ou non, être un arrondi fidèle.

Preuve du lemme 7.1. Puisque $\widehat{c} = \mathsf{Horner}(p_\pi \oplus p_\sigma, x)$, avec p_π et p_σ deux polynômes à coefficients de degré $n-1$, d'après le lemme 4.5, on a

$$|\widehat{c} - c| \leq \gamma_{2n-1}(\widetilde{p_\pi + p_\sigma})(x).$$

D'autre part, un calcul simple, ne faisant intervenir que le modèle standard de l'arithmétique flottante, montre que

$$(\widetilde{p_\pi + p_\sigma})(x) \leq (1+\mathbf{u})^{2n-1}\mathsf{Horner}(|p_\pi + p_\sigma|, |x|).$$

En notant $b := \mathsf{Horner}(p_\pi \oplus p_\sigma, x)$, on a donc

$$|\widehat{c} - c| \leq (1+\mathbf{u})^{2n-1}\gamma_{2n-1}b.$$

Comme $\gamma_{2n-1} \leq (1+\mathbf{u})\widehat{\gamma}_{2n-1}$, il vient

$$|\widehat{c} - c| \leq (1+\mathbf{u})^{2n}\widehat{\gamma}_{2n-1}b \leq (1+\mathbf{u})^{2n+1}\widehat{\gamma}_{2n-1} \otimes b.$$

Rappelons ici le relation (2.8). Si $a \in \mathbf{F}$ et $k \in \mathbf{N}$, avec $(k+1)\mathbf{u} < 1$, alors en l'absence d'underflow on a

$$(1+\mathbf{u})^k|a| \leq \mathrm{fl}\left(\frac{|a|}{1-(k+1)\mathbf{u}}\right).$$

D'où finalement $|\widehat{c} - c| \leq (\widehat{\gamma}_{2n-1} \otimes b) \oslash (1 - 2(n+1)\mathbf{u})$. ■

7.2.2 Borne d'erreur *a posteriori* pour le résultat compensé

Nous déduisons maintenant du lemme 7.1 une borne *a posteriori* sur l'erreur entachant le résultat compensé $\overline{r}$. Dans la preuve du théorème 4.4, nous avons fait apparaître la relation (4.12), que nous rappelons ici :

$$|\overline{r} - p(x)| \leq \mathbf{u}|p(x)| + (1+\mathbf{u})|\widehat{c} - c|.$$

Cette relation permet de majorer l'erreur $|\overline{r} - p(x)|$ entachant le résultat compensé $\overline{r}$ en fonction de l'erreur $|\widehat{c} - c|$ commise lors du calcul du terme correctif. Mais elle fait intervenir le résultat exact $p(x)$ de l'évaluation, ce qui la rend inutilisable en pratique. Afin de trouver une borne qui puisse être calculée en arithmétique flottante, il nous faut obtenir une majoration similaire, mais ne faisant plus intervenir $p(x)$. On écrit

$$|\overline{r} - p(x)| = |(\widehat{r} \oplus \widehat{c}) - p(x)| \leq |(\widehat{r} \oplus \widehat{c}) - (\widehat{r} + \widehat{c})| + |(\widehat{r} + \widehat{c}) - p(x)|.$$

On définit ici δ comme étant l'erreur d'arrondi commise lors de l'addition $\widehat{r} \oplus \widehat{c}$. D'après le théorème 3.6,

$$\delta := (\widehat{r} + \widehat{c}) - (\widehat{r} \oplus \widehat{c}) \in \mathbf{F}.$$

D'autre part, comme dans la preuve du théorème 4.4, puisque EFTHorner est une transformation exacte, on a $\widehat{r} = p(x) - c$, donc $|\overline{r} - p(x)| \leq |\delta| + |\widehat{c} - c|$. On obtient ainsi

$$|\overline{r} - p(x)| \leq (|\delta| \oplus |\widehat{c} - c|) \oslash (1 - 2u). \tag{7.3}$$

L'erreur directe $|\widehat{c} - c|$ est facilement majorée à l'aide du lemme 7.1.

Le terme $|\delta|$ dans (7.3) peut être majoré par $\mathbf{u}|\overline{r}|$ en utilisant uniquement le modèle standard l'arithmétique flottante. Mais lorsque le conditionnement est petit devant $\mathbf{u}^{-1}$, les expériences numériques du chapitre 4 (Section 4.5) montrent que l'erreur relative qui entache le résultat calculé par CompHorner peut être inférieure à l'unité d'arrondi $\mathbf{u}$: en d'autres termes, l'erreur absolue $|\overline{r} - p(x)|$ peut dans ce cas être inférieure à $\mathbf{u}|\overline{r}|$. Mais rappelons que δ est l'erreur d'arrondi commise lors de l'addition $\overline{r} = \widehat{r} \oplus \widehat{c}$. Le terme $|\delta|$ peut ainsi être calculé à l'aide de l'algorithme TwoSum (algorithme 3.5). Par la suite, nous préférerons donc toujours calculer $|\delta|$ exactement, plutôt que de majorer ce terme par $\mathbf{u}|\overline{r}|$.

En combinant la majoration de l'erreur entachant le terme correctif calculé proposée dans le lemme 7.1, et la relation (7.3), on obtient le théorème 7.2.

Théorème 7.2. *On considère un polynôme p de degré n à coefficients flottants, et on utilise les notations de l'algorithme 4.3. Soit $\alpha \in \mathbf{F}$, le majorant de $|\widehat{c} - c|$ défini par la relation (7.2). On suppose que $2(n+1)\mathbf{u} < 1$, et qu'aucun underflow n'intervient.*

- *Si $\alpha < \frac{\mathbf{u}}{2}|\overline{r}|$, alors $\overline{r}$ est un arrondi fidèle de $p(x)$.*
- *Soit $\delta \in \mathbf{F}$, l'erreur d'arrondi commise lors de l'addition $\widehat{r} \oplus \widehat{c}$, i.e., $\overline{r} + \delta = \widehat{r} + \widehat{c}$. L'erreur directe entachant le résultat $\overline{r} = \mathsf{CompHorner}(p, x)$ est majorée comme suit,*

$$|\overline{r} - p(x)| \leq (|\delta| \oplus \alpha) \oslash (1 - 2\mathbf{u}) =: \beta.$$

D'après le théorème 7.2, on déduit l'algorithme CompHornerBound ci-dessous, qui calcule le résultat compensé $\overline{r} = \mathsf{CompHorner}(p, x)$, tout en renvoyant une borne d'erreur β telle que $|\overline{r} - p(x)| \leq \beta$.

Algorithme 7.3. Schéma de Horner compensé, avec calcul d'une borne d'erreur et test de l'arrondi fidèle.

function $[\overline{r}, \beta, \mathsf{isfaith}] = \mathsf{CompHornerBound}(p, x)$
  if $2(n+1)\mathbf{u} \geq 1$, error('Validation impossible'), end
  $[\widehat{r}, p_\pi, p_\sigma] = \mathsf{EFTHorner}(p, x)$
  $\widehat{c} = \mathsf{Horner}(p_\pi \oplus p_\sigma, x)$
  $[\overline{r}, \delta] = \mathsf{TwoSum}(\widehat{r}, \widehat{c})$
  $b = \mathsf{Horner}(|p_\pi \oplus p_\sigma|, |x|)$
  $\alpha = (\widehat{\gamma}_{2n-1} \otimes b) \oslash (1 - 2(n+1)\mathbf{u})$
  $\beta = (|\delta| \oplus \alpha) \oslash (1 - 2\mathbf{u})$
  $\mathsf{isfaith} = \left(\alpha < \frac{\mathbf{u}}{2}|\overline{r}|\right)$

De plus, CompHornerBound effectue un test pour déterminer si le résultat compensé est un arrondi fidèle du résultat exact : si le booléen isfaith retourné par $\mathsf{CompHornerBound}(p, x)$ a pour valeur « vrai », alors on est assuré que $\overline{r}$ est un arrondi fidèle de $p(x)$. Dans le cas contraire, on ne peut rien affirmer : $\overline{r}$ peut, ou non, être un arrondi fidèle de $p(x)$.

7.3 Expériences numériques

Tous nos tests sont effectués sous MATLAB, en double précision IEEE-754. Les expériences rapportées ci-dessous visent à illustrer la finesse de la borne *a posteriori* proposée dans la section précédente, ainsi que celle du test *a posteriori* de l'arrondi fidèle.

7.3.1 Finesse de la borne d'erreur *a posteriori*

Nous comparons ici la finesse de la borne d'erreur *a posteriori* β, fournie par l'algorithme 7.3, avec :

- celle de la borne *a priori* (4.13)
- l'erreur directe mesurée.

Pour cela, nous évaluons le polynôme $p_5(x) = (1-x)^5$ sous sa forme développée, en 512 points au voisinage de $x = 1$. Les résultats de ce test sont reportés sur la figure 7.1.

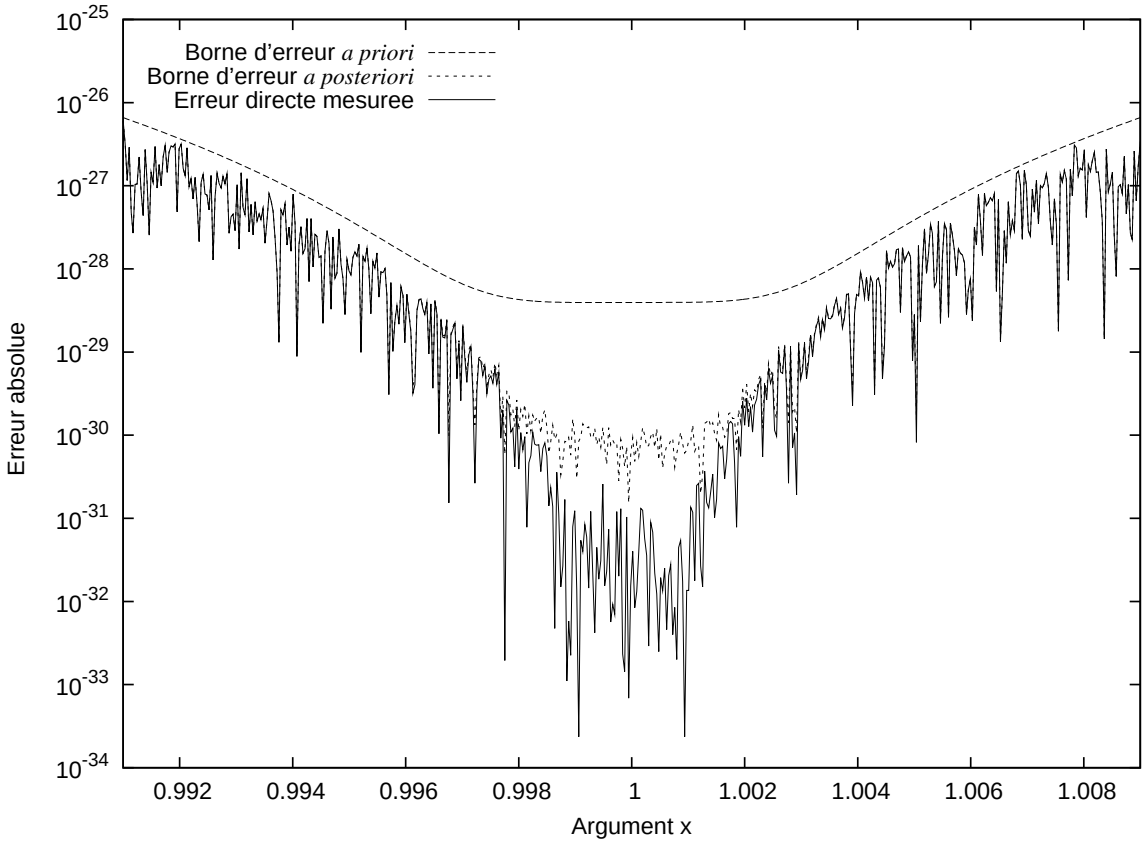


FIG. 7.1 – Comparaison de la borne d'erreur *a posteriori* pour CompHorner, avec la borne d'erreur *a priori* et l'erreur directe mesurée.

Comme dans nos expériences précédentes, on constate que la borne *a priori* ne se confond jamais, sur l'intervalle considéré, avec l'erreur mesurée. En outre, plus x est proche de la racine 1, c'est à dire plus le conditionnement est élevé, moins la borne *a priori* est fine. D'autre part, on constate que lorsque x est suffisamment éloigné de la racine, il est difficile de différencier la borne d'erreur *a posteriori* de l'erreur effective mesurée. La borne *a posteriori* est également bien plus fine que la borne *a priori* au voisinage de 1.

7.3.2 Arrondi fidèle

Nous nous intéressons ici à la fois au critère *a priori* du théorème 4.10 et au test *a posteriori* effectué par l'algorithme 7.3. Rappelons que deux cas peuvent se produire lorsque ce test *a posteriori* est exécuté.

1. Si le test est satisfait, c'est à dire si la valeur booléenne isfaith retournée par l'algorithme CompHornerBound est « vrai », cela prouve que le résultat compensé $\overline{r}$ est un arrondi fidèle de $p(x)$. Ce cas sera représenté sur nos graphiques à l'aide d'un petit carré ($\square$).
2. Si le test échoue, alors le résultat compensé peut, ou non, être un arrondi fidèle de $p(x)$. Nous distinguons alors deux nouveaux sous-cas :
   (a) Si $\overline{r}$ est effectivement un arrondi fidèle de $p(x)$, nous représenterons cette situation par un petit disque $\bullet$.
   (b) Sinon, le résultat compensé n'est pas un arrondi fidèle, ce que nous représenterons par une croix $\times$.

Dans un premier temps, nous utilisons à nouveau les polynômes $p_n(x) = (1-x)^n$, pour les degrés $n = 6$ et $n = 8$. Les résultats de ces expériences sont représentés sur la figure 7.2, où l'on évalue les polynômes p_6 et p_8 au voisinage de 1. Nous représentons également l'évolution du nombre de conditionnement en fonction de x sur le voisinage considéré.

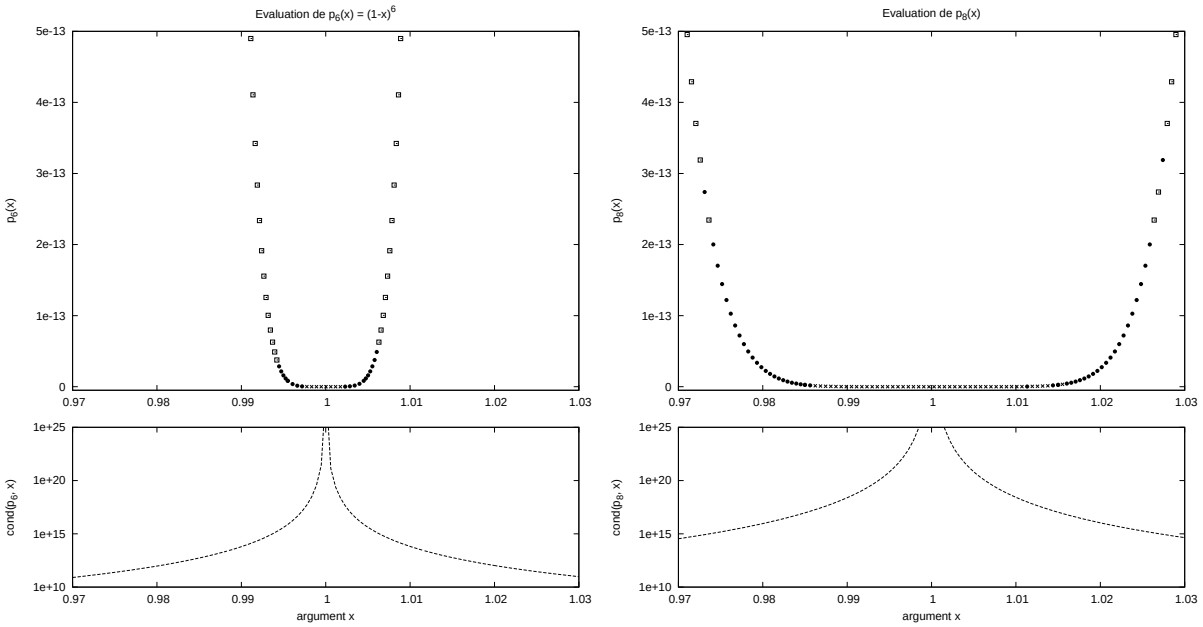


FIG. 7.2 – Test de CompHornerBound à l'aide des polynômes $p_n(x) = (1-x)^n$, pour $n = 6, 8$, sous forme développée, évalués au voisinage de $x = 1$.

Ces expériences numériques illustrent bien la relation entre la perte de précision de l'évaluation et la proximité d'une racine multiple, c'est à dire l'augmentation du nombre de conditionnement. Pour les évaluations effectuées suffisamment loin de la racine 1, le test *a posteriori* de l'arrondi fidèle est effectivement satisfait. On constate également qu'il existe une large zone, de part et d'autre de la racine, dans laquelle les évaluations compensées sont fidèles, mais où le test échoue.

Nous effectuons aussi des tests à l'aide de polynômes générés aléatoirement : nous utilisons ici le même jeu de polynômes que celui utilisé dans les expériences numériques de la Section 4.5. Pour chaque évaluation effectuée, nous reportons sur la figure 7.3 la précision relative en fonction du nombre de conditionnement. La borne *a priori* sur le conditionnement, définie par l'inégalité (4.14) et permettant d'assurer une évaluation fidèle, est également représentée.

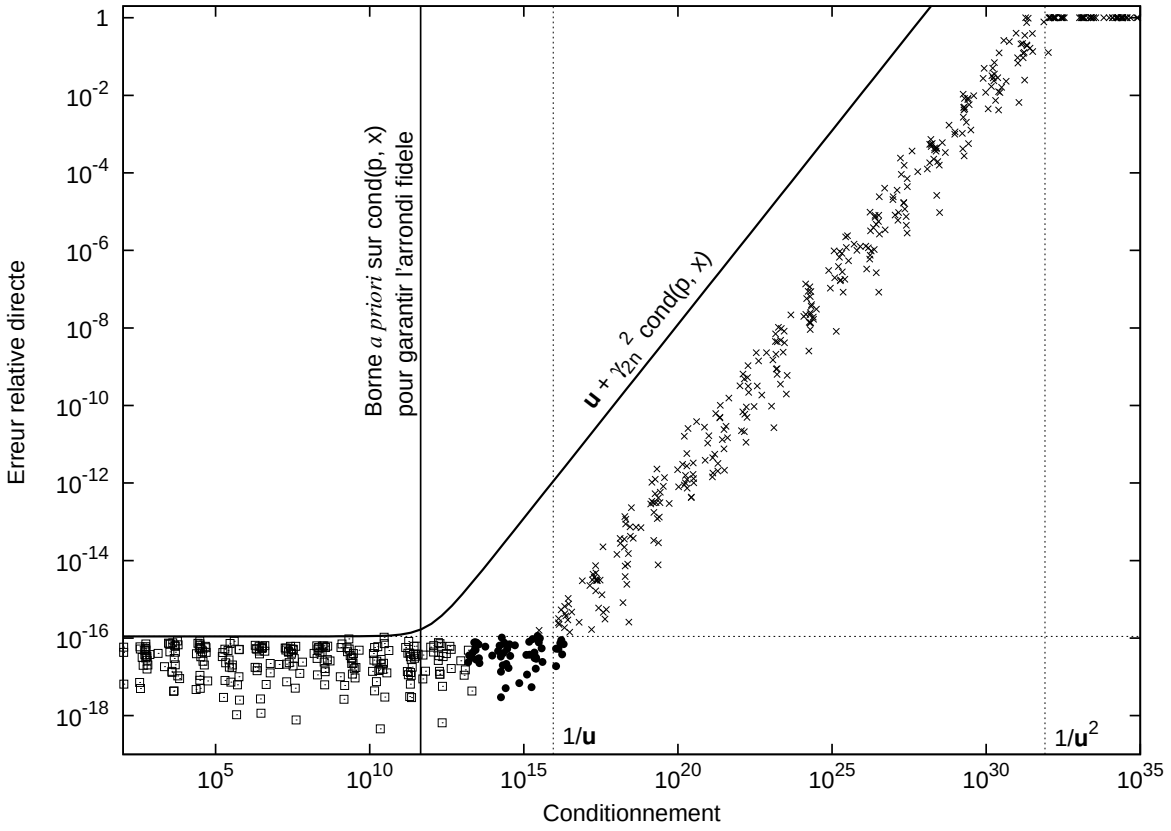


FIG. 7.3 – Arrondi fidèle avec le schéma de Horner compensé (polynômes de degré 50).

On constate que toutes les évaluations compensées dont le nombre de conditionnement est inférieur à cette borne *a priori* sont effectivement fidèles. On constate également que le test *a posteriori* est satisfait pour des évaluations dont le nombre de conditionnement est supérieur à la borne (4.14) – rappelons que les évaluations prouvées fidèles par le test *a posteriori* sont représentées par un petit carré. On notera finalement que le schéma de Horner compensé produit en pratique un arrondi fidèle du résultat exact pour des nombres de conditionnement atteignant $\mathbf{u}^{-1}$ – évaluations reportées à l'aide d'un petit disque.

7.4 Performances de CompHornerBound

Dans cette section, nous montrons comment implanter efficacement l'algorithme validé CompHornerBound (algorithme 7.3) : nous montrons comment implanter l'algorithme à l'aide d'une boucle simple, en ne parcourant qu'une fois les coefficients du polynôme à évaluer. Nous détaillons ensuite les tests de performance que nous avons effectués, et qui démontrent l'efficacité pratique de cet algorithme en terme de temps de calcul.

7.4.1 Implantation pratique de l'algorithme

Supposons dans un premier temps que l'architecture cible ne dispose pas de l'instruction FMA. Il est facile de voir que l'algorithme CompHornerBound peut être implanté à l'aide d'une boucle simple, en ne parcourant qu'une seule fois les coefficients du polynôme $p(x) = \sum_{i=0}^{n} a_i x^i$.

Il suffit de procéder comme cela a été expliqué dans le cas du schéma de Horner compensé à la Section 5.2. En particulier, le découpage de x n'est effectué qu'une seule fois avant le début de la boucle : l'instruction $\mathsf{TwoProd}(r_{i+1}, x)$ n'engendre ainsi que 13 opérations flottantes, au lieu des 17 nécessitées classiquement. Cela nous donne l'algorithme 7.4 suivant.

Algorithme 7.4. Implantation pratique de CompHornerBound.

function $[\overline{r}, \beta, \mathsf{isfaith}] = \mathsf{CompHornerBound}(p, x)$
  if $2(n+1)\mathbf{u} \geq 1$, error('Validation impossible'), end
  $[xh, xl] = \mathsf{Split}(x)$
  $r_n = a_n$; $b_n = c_n = 0$
  for $i = n-1 : -1 : 0$
    $[p_i, \pi_i] = \mathsf{TwoProd}_{xh,xl}(r_{i+1}, x)$
    $[r_i, \sigma_i] = \mathsf{TwoSum}(p_i, a_i)$
    $t = \pi_i \oplus \sigma_i$
    $c_i = c_{i+1} \otimes x \oplus t$
    $b_i = b_{i+1} \otimes |x| \oplus |t|$
  end
  $[\overline{r}, \delta] = \mathsf{TwoSum}(r_0, c_0)$
  $\alpha = (\widehat{\gamma}_{2n-1} \otimes b_0) \oslash (1 - 2(n+1)\mathbf{u})$
  $\beta = (|\delta| \oplus \alpha) \oslash (1 - 2\mathbf{u})$
  $\mathsf{isfaith} = \left(\alpha < \frac{\mathbf{u}}{2}|\overline{r}|\right)$

En l'absence de FMA, on peut donc constater que l'algorithme CompHornerBound nécessite $24n + \mathcal{O}(1)$ opérations flottantes, contre $22n + \mathcal{O}(1)$ pour le schéma de Horner compensé (algorithme 4.3).

Nous ne décrivons pas en détail l'étude des améliorations qu'il est possible d'apporter à l'algorithme CompHornerBound lorsque l'instruction FMA est disponible, comme nous l'avons fait au chapitre 6 pour le schéma de Horner compensé. Précisons néanmoins que dans les tests de performance reportés ci-dessous, nous utilisons effectivement une version adaptée de l'algorithme CompHornerBound : cet algorithme est obtenu en procédant à la validation de l'algorithme $\mathsf{CompHorner}_{\mathsf{fma}}$ (algorithme 6.3). Le coût théorique de l'algorithme validé est de $11n + \mathcal{O}(1)$ opérations flottantes, contre $10n + \mathcal{O}(1)$ pour l'algorithme $\mathsf{CompHorner}_{\mathsf{fma}}$.

7.4.2 Tests de performances

Nous réalisons nos mesures dans les environnements (I) à (VI) listés dans le tableau 5.1. Les mesures sont effectuées de la même manière que celles réalisées au chapitre 5 : nous utilisons un jeu de 39 polynômes, dont le degré varie de 10 à 200 par pas de 5. La double précision IEEE-754 est utilisée comme précision de travail. Pour chaque degré, nous mesurons :

- le ratio du temps d'exécution de CompHornerBound sur celui de CompHorner, ce qui quantifie le surcoût introduit par la validation ;
- le ratio du temps d'exécution de CompHornerBound sur celui du schéma de Horner, afin de quantifier le surcoût introduit par la méthode validée par rapport à l'algorithme que nous utilisons comme référence.

Nous reportons dans le tableau 7.1 la moyenne des ratios mesurés dans chaque environnement.

TAB. 7.1 – Surcoût moyen mesuré pour CompHorner, DDHorner et CompHornerBound.

environnement		$\frac{\text{CompHornerBound}}{\text{CompHorner}}$	$\frac{\text{CompHornerBound}}{\text{Horner}}$	$\frac{\text{CompHorner}}{\text{Horner}}$	$\frac{\text{DDHorner}}{\text{Horner}}$
(I)	P4, gcc, x87	1.2	3.5	2.8	8.6
(II)	P4, gcc, sse	1.3	3.9	3.1	8.9
(III)	P4, icc, x87	1.2	3.2	2.7	9.0
(IV)	P4, icc, sse	1.2	3.9	3.3	9.8
(V)	Ath64, gcc, sse	1.1	3.6	3.2	8.7
(VI)	Itanium, gcc	1.2	3.4	2.8	6.7
(VII)	Itanium, icc	1.1	1.7	1.5	5.9

Pour mémoire, nous reportons également dans le tableau 7.1 la moyenne des surcoûts mesurés pour les algorithmes CompHorner et DDHorner. Rappelons que ces mesures ont déjà été présentées aux chapitres 5 et 6.

Dans chacun des environnements considérés ici, on constate que le surcoût introduit par la validation *a posteriori* est très faible. D'autre part, les tests de performance des chapitres 4 et 6 nous ont permis de conclure que l'évaluation polynomiale compensée s'exécute au moins deux fois plus rapidement que le schéma de Horner en arithmétique double-double. Le tableau 7.1 nous permet également de constater que CompHornerBound s'exécute environ deux fois plus rapidement que DDHorner.

On retiendra donc qu'il est environ deux fois moins coûteux de calculer une évaluation validée en précision doublée à l'aide de CompHornerBound, que de calculer un résultat de précision similaire, mais non validé, en arithmétique double-double.

Il est à noter que l'étude menée au chapitre 5 pour expliquer les performances de CompHorner permet également d'expliquer celles de CompHornerBound. En effet, la validation n'ajoute que deux opérations arithmétiques à l'itération effectuée par CompHorner (ligne $b_i = b_{i+1} \otimes |x| \oplus |t|$ dans l'algorithme 7.4 ci-dessus). De plus, il est aisé de constater ces deux opérations peuvent s'exécuter en concurrence avec les autres instructions d'une itération. Cela explique le faible surcoût de la validation de l'évaluation compensée mesurée sur les architectures modernes, ainsi que les performances de CompHornerBound face à DDHorner.

7.5 Validation en présence d'underflow

Les résultats obtenus à la Section 7.2 sont valides tant qu'aucun underflow n'intervient au cours des calculs. Dans cette section, nous nous concentrons sur le problème du calcul d'une borne d'erreur *a posteriori* pour le résultat compensé $\overline{r} = \mathsf{CompHorner}(p, x)$, valide

également en présence d'underflow — nous ne nous intéresserons plus au problème du test *a posteriori* de l'arrondi fidèle.

Comme dans la Section 4.6, il nous faut supposer que chaque multiplication apparaissant dans le calcul de $\overline{r}$ est susceptible de produire un résultat dénormalisé. Mais il nous faut aussi envisager la possibilité d'un underflow pour chaque multiplication intervenant dans le calcul de la borne d'erreur. Cela introduit dans nos preuves des termes multiples de l'unité d'underflow $\mathbf{v}$, sur lesquels il faut travailler de manière à ce que l'expression obtenue fournisse bien, lorsqu'elle sera évaluée en arithmétique flottante, un majorant de $|\overline{r} - p(x)|$.

7.5.1 Notations et relations utiles

On utilise bien entendu les mêmes notations que dans la Section 4.12. Le mode d'arrondi courant est l'arrondi au plus proche. Rappelons que l'on désigne par

- λ le plus petit flottant positif normalisé,
- $\mathbf{v} = 2\lambda\mathbf{u}$ l'unité d'underflow, c'est à dire le plus petit flottant positif dénormalisé.

Soit $\circ \in \{+, -, \times, /\}$ une opération élémentaire exacte, et soient $a, b \in \mathbf{F}$. Pour les preuves du lemme 7.6 et du théorème 7.8, nous utiliserons essentiellement les inégalités suivantes,

$$\mathrm{fl}(|a \circ b|) \leq (1 + \mathbf{u})|a \circ b| + \eta,$$

et

$$(1 - \mathbf{u})(|a| \circ |b|) \leq \mathrm{fl}(|a| \circ |b|) + \eta,$$

avec $\eta = 0$ si $\circ \in \{+, -\}$, et $|\eta| \leq \mathbf{v}$ sinon.

Dans la preuve du théorème 7.8, nous utiliserons le lemme suivant.

Lemme 7.5. *On suppose que l'on travaille dans le mode d'arrondi au plus proche. Soit k un entier naturel. Si $k\mathbf{u} \leq 1$, alors $k\mathbf{v} \in \mathbf{F}$.*

Preuve. Les flottants compris entre 0 et 2λ sont régulièrement espacés [39, p.37] : l'espace entre deux dénormalisés consécutifs est $\mathbf{v}$. Autrement dit, pour tout $a \in \mathbf{F}$ tel que $0 \leq a \leq 2\lambda$, il existe $k \in \mathbf{N}$ tel que $a = k\mathbf{v}$. Or, $0 \leq k\mathbf{v} \leq 2\lambda$ si et seulement si $0 \leq k\mathbf{u} \leq 1$. ■

7.5.2 Évaluation de la somme de deux polynômes

Soient $p(x) = \sum_{i=0}^{n} a_i x^i$ et $q(x) = \sum_{i=0}^{n} b_i x^i$ deux polynômes à coefficients flottants, et soit x un argument flottant. On définit les réels r_i par

$$r_n = |a_n + b_n| \quad \text{et} \quad r_i = r_{i+1}|x| + |a_i + b_i|, \quad \text{pour} \quad i = n-1, \ldots, 0.$$

Définissons les $\widehat{r}_i$ comme l'évaluation en arithmétique flottante des r_i, c'est à dire

$$\widehat{r}_n = |a_n \oplus b_n| \quad \text{et} \quad \widehat{r}_i = \widehat{r}_{i+1} \otimes |x| \oplus |a_i \oplus b_i| \quad \text{pour} \quad i = n-1, \ldots, 0.$$

On a en particulier $r_0 = (\widetilde{p+q})(x)$ et $\widehat{r}_0 = \mathsf{Horner}(|p \oplus q|, |x|)$.

Lemme 7.6. *Même en présence d'underflow, on a*

$$(\widetilde{p+q})(x) \leq (1+\mathbf{u})^{2n+1}\mathsf{Horner}(|p \oplus q|, |x|) + (1+\mathbf{u})^{2n-1}\mathbf{v}\sum_{i=0}^{n-1}|x|^i.$$

Preuve. On a $r_n \leq (1+\mathbf{u})\widehat{r}_n$. De plus, $r_{n-1} = r_n|x| + |a_{n-1} + b_{n-1}|$, d'où

$$\begin{aligned} r_{n-1} &\leq (1+\mathbf{u})\widehat{r}_n|x| + |a_{n-1} + b_{n-1}| \\ &\leq (1+\mathbf{u})^2\widehat{r}_n \otimes |x| + (1+\mathbf{u})|a_{n-1} \oplus b_{n-1}| + (1+\mathbf{u})\mathbf{v} \\ &\leq (1+\mathbf{u})^3\widehat{r}_{n-1} + (1+\mathbf{u})\mathbf{v}. \end{aligned}$$

À l'étape suivante, $r_{n-2} = r_{n-1}|x| + |a_{n-2} + b_{n-2}|$, donc

$$\begin{aligned} r_{n-2} &\leq (1+\mathbf{u})^3\widehat{r}_{n-1}|x| + (1+\mathbf{u})\mathbf{v}|x| + |a_{n-2} + b_{n-2}| \\ &\leq (1+\mathbf{u})^4\widehat{r}_{n-1} \otimes |x| + (1+\mathbf{u})^3\mathbf{v} + (1+\mathbf{u})\mathbf{v}|x| + (1+\mathbf{u})|a_{n-1} \oplus b_{n-1}| \\ &\leq (1+\mathbf{u})^5(\widehat{r}_{n-1} \otimes |x| \oplus |a_{n-1} \oplus b_{n-1}|) + (1+\mathbf{u})^3\mathbf{v}(1+|x|) \\ &\leq (1+\mathbf{u})^5\widehat{r}_{n-2} + (1+\mathbf{u})^3\mathbf{v}(1+|x|). \end{aligned}$$

On peut montrer par récurrence que

$$r_{n-i} \leq (1+\mathbf{u})^{2i+1}\widehat{r}_{n-i} + (1+\mathbf{u})^{2i-1}\mathbf{v}\sum_{j=0}^{i-1}|x|^j,$$

d'où le résultat annoncé. ■

En combinant le lemme 4.13 avec le résultat précédent, et comme $\gamma_{2n+1}(1+\mathbf{u})^{2n+1} \leq \gamma_{4n+2}$ et $\gamma_{2n-1} + \gamma_{2n+1}(1+\mathbf{u})^{2n-1} \leq \gamma_{6n-1}$, on obtient le lemme suivant.

Lemme 7.7. *Même en présence d'underflow, l'erreur commise en évaluant la somme de deux polynômes par le schéma de Horner est majorée comme suit,*

$$|\mathsf{Horner}(p \oplus q, x) - (p+q)(x)| \leq \gamma_{4n+2}\mathsf{Horner}(|p \oplus q|, |x|) + (1+\gamma_{6n-1})\mathbf{v}\sum_{i=0}^{n-1}|x|^i.$$

7.5.3 Borne d'erreur *a posteriori* pour le résultat compensé

Le théorème suivant fournit une borne d'erreur *a posteriori*, valide également en présence d'underflow, pour le résultat compensé $\overline{r}$.

Théorème 7.8. *On considère un polynôme p de degré n à coefficients flottants, et on utilise les notations de l'algorithme 4.3. On définit les deux constantes $\mu = 14\mathbf{v} \in \mathbf{F}$ et $\nu = 6\mathbf{v} \in \mathbf{F}$, et on suppose $14n\mathbf{u} \leq 1$. Soient également :*

- *$\delta \in \mathbf{F}$ l'erreur d'arrondi commise lors de l'addition $\widehat{r} \oplus \widehat{c}$, i.e., $\overline{r} + \delta = \widehat{r} + \widehat{c}$,*
- *$b \in \mathbf{F}$ tel que $b = \mathsf{Horner}(|p_\pi \oplus p_\sigma|, |x|)$,*
- *$\rho \in \mathbf{F}$ tel que $\max(1, |x|^{n-1}) \leq \rho$.*

Alors, l'erreur directe entachant le résultat compensé $\overline{r} = \mathsf{CompHorner}(p, x)$ est majorée comme suit,

$$|\overline{r} - p(x)| \leq |\delta| \oplus (\widehat{\gamma}_{4n+2} \otimes b \oplus (2\mathbf{u} \otimes |\delta| \oplus (n\mu \otimes \rho \oplus \nu))).$$

Preuve. On débute la preuve comme dans le cas « sans underflow » de la Section 7.2 :

$$|\overline{r} - p(x)| = |(\widehat{r} \oplus \widehat{c}) - p(x)| \leq |(\widehat{r} \oplus \widehat{c}) - (\widehat{r} + \widehat{c})| + |(\widehat{r} + \widehat{c}) - p(x)|.$$

Rappelons que δ est l'erreur d'arrondi commise lors de l'addition $\widehat{r} \oplus \widehat{c}$. On a donc $\delta = (\widehat{r} + \widehat{c}) - (\widehat{r} \oplus \widehat{c}) \in \mathbf{F}$, et $|\overline{r} - p(x)| \leq |\delta| + |\widehat{r} + \widehat{c} - p(x)|$. Ici, il est important de noter que EFTHorner n'étant plus une transformation exacte en présence d'underflow, $p(x) \neq \widehat{r} + c$. Néanmoins, d'après le théorème 4.14, et comme $c = (p_\pi + p_\sigma)(x)$, on écrit

$$p(x) = \widehat{r} + c + 5\sum_{i=0}^{n-1} \eta_i x^i, \quad \text{avec} \quad |\eta_i| \leq \mathbf{v}.$$

Il vient

$$|\overline{r} - p(x)| \leq |\delta| + |c - \widehat{c}| + 5n\mathbf{v}\rho.$$

Comme $c = (p_\pi + p_\sigma)(x)$ et $\widehat{c} = \mathsf{Horner}(p_\pi \oplus p_\sigma, x)$, avec p_π et p_σ deux polynômes de degré $n-1$, d'après le lemme 7.7,

$$\begin{aligned} |c - \widehat{c}| &\leq \gamma_{4n-2}\mathsf{Horner}(|p_\pi \oplus p_\sigma|, |x|) + n(1+\gamma_{6n-7})\mathbf{v}\max(1, |x|^{n-2}) \\ &\leq \gamma_{4n-2}b + n(1+\gamma_{6n-7})\mathbf{v}\max(1, |x|^{n-2}). \end{aligned}$$

Comme $14n\mathbf{u} \leq 1$, on a également $2(6n-7)\mathbf{u} \leq 1$, donc $\gamma_{6n-7} \leq 1$. En majorant de plus $\max(1, |x|^{n-2})$ par ρ, on obtient $|c - \widehat{c}| \leq \gamma_{4n-2}b + 2n\mathbf{v}\rho$. On a donc

$$|\overline{r} - p(x)| \leq |\delta| + \gamma_{4n-2}b + 7n\mathbf{v}\rho.$$

Reste à transformer l'inégalité ci-dessus, de manière à obtenir une expression qui puisse être calculée en arithmétique flottante. Pour simplifier l'exercice, nous utiliserons ci-dessous la notation fl(.) pour désigner les sous-expressions évaluées en arithmétique flottante : on supposera que les opérations sont effectuées de gauche à droite, et dans l'ordre indiqué par les parenthèses. Puisque $\gamma_{4n-2} \leq (1-\mathbf{u})^4\gamma_{4n+2} \leq (1-\mathbf{u})^3\widehat{\gamma}_{4n+2}$, on a,

$$\begin{aligned} |\overline{r} - p(x)| &\leq (1-\mathbf{u})|\delta| + (1-\mathbf{u})^3\widehat{\gamma}_{4n+2}b + \mathbf{u}|\delta| + 7n\mathbf{v}\rho \\ &\leq (1-\mathbf{u})|\delta| + (1-\mathbf{u})^2\,\mathrm{fl}(\widehat{\gamma}_{4n+2}b) + \mathbf{u}|\delta| + 7n\mathbf{v}\rho + \mathbf{v}. \end{aligned}$$

Comme $2(1-\mathbf{u})^3 \geq 1$, on a $\mathbf{u}|\delta| \leq (1-\mathbf{u})^3 2\mathbf{u}|\delta|$, et

$$|\overline{r} - p(x)| \leq (1-\mathbf{u})|\delta| + (1-\mathbf{u})^2\,\mathrm{fl}(\widehat{\gamma}_{4n+2}b) + (1-\mathbf{u})^3\,\mathrm{fl}(2\mathbf{u}|\delta|) + 7n\mathbf{v}\rho + 2\mathbf{v}.$$

De même, $2(1-\mathbf{u})^5 \geq 1$, donc $7n\mathbf{v}\rho \leq (1-\mathbf{u})^5 14n\mathbf{v}\rho$, et

$$|\overline{r} - p(x)| \leq (1-\mathbf{u})|\delta| + (1-\mathbf{u})^2\,\mathrm{fl}(\widehat{\gamma}_{4n+2}b) + (1-\mathbf{u})^3\,\mathrm{fl}(2\mathbf{u}|\delta|) + (1-\mathbf{u})^5 14n\mathbf{v}\rho + 2\mathbf{v}.$$

Puisque par hypothèse $14n\mathbf{u} \leq 1$, d'après le lemme 7.5 on a $14n\mathbf{v} = n\mu \in \mathbf{F}$. Par contre, le résultat de l'opération $\mathrm{fl}(n\mu\rho)$ est susceptible d'être dénormalisé, avec perte de précision. On écrit donc

$$\begin{aligned} (1-\mathbf{u})^5 14n\mathbf{v}\rho + 2\mathbf{v} &\leq (1-\mathbf{u})^4\,\mathrm{fl}(n\mu\rho) + 3\mathbf{v} \\ &\leq (1-\mathbf{u})^4\,\mathrm{fl}(n\mu\rho) + (1-\mathbf{u})^4 6\mathbf{v} \\ &\leq (1-\mathbf{u})^4\,\mathrm{fl}(n\mu\rho) + (1-\mathbf{u})^4\nu \\ &\leq (1-\mathbf{u})^3\,\mathrm{fl}(n\mu\rho + \nu). \end{aligned}$$

On obtient finalement

$$\begin{aligned}
|\overline{r} - p(x)| &\leq (1-\mathbf{u})|\delta| + (1-\mathbf{u})^2\,\mathrm{fl}(\widehat{\gamma}_{4n+2}b) + (1-\mathbf{u})^3\big(\mathrm{fl}(2\mathbf{u}|\delta|) + \mathrm{fl}(n\mu\rho+\nu)\big)\\
&\leq (1-\mathbf{u})|\delta| + (1-\mathbf{u})^2\big(\mathrm{fl}(\widehat{\gamma}_{4n+2}b) + \mathrm{fl}(2\mathbf{u}|\delta| + (n\mu\rho+\nu))\big)\\
&\leq (1-\mathbf{u})\big(|\delta| + \mathrm{fl}(\widehat{\gamma}_{4n+2}b + (2\mathbf{u}|\delta| + (n\mu\rho+\nu)))\big)\\
&\leq \mathrm{fl}(|\delta| + (\widehat{\gamma}_{4n+2}b + (2\mathbf{u}|\delta| + (n\mu\rho+\nu)))).
\end{aligned}$$

Ceci achève la preuve du théorème 7.8. ■

Lorsque $|x| \leq 1$, le calcul d'un majorant pour $\max(1, |x|^{n-1})$ est bien entendu trivial. Dans le cas $|x| > 1$, nous utiliserons le lemme suivant.

Lemme 7.9. *Soit k un entier naturel, et soit $x \in \mathbf{F}$ tel que $|x| > 1$. Soit $\mathrm{fl}(|x|^k)$ une valeur approchée de $|x|^k$ calculée en arithmétique flottante à la précision $\mathbf{u}$, en n'effectuant que des produits flottants, dans un ordre quelconque. Alors*

$$|x|^k \leq (1+\mathbf{u})^{k-1}\,\mathrm{fl}(|x|^k) \leq \mathrm{fl}(|x|^k) \oslash (1-(k+1)\mathbf{u}).$$

Preuve. Comme $|x| > 1$, aucun résultat dénormalisé n'intervient au cours du calcul de $\mathrm{fl}(|x|^k)$. La méthode de calcul de $\mathrm{fl}(|x|^k)$ qui conduit au plus grand nombre de multiplications est la méthode naïve qui consiste à multiplier $k-1$ fois $|x|$ par lui-même : $k-1$ produits flottants sont alors effectués. Dans ce cas, d'après le modèle standard de l'arithmétique flottante, il est facile de voir que $|x|^k \leq (1+\mathbf{u})^{k-1}\,\mathrm{fl}(|x|^k)$. ■

D'après le théorème 7.8, et en utilisant le lemme 7.9 pour le calcul de ρ, on obtient l'algorithme suivant, qui calcule le résultat compensé $\overline{r} = \mathsf{CompHorner}(p, x)$, ainsi qu'une borne β sur l'erreur directe $|\overline{r} - p(x)|$.

Algorithme 7.10. Schéma de Horner compensé, avec calcule d'une borne d'erreur valide en présence d'underflow.

function $[\overline{r}, \beta] = \mathsf{CompHornerBound}(p, x)$
  if $7n\mathbf{u} > 1$, error('Validation impossible'), end
  $r_n = a_n$; $b_n = c_n = 0$
  for $i = n-1 : -1 : 0$
    $[p_i, \pi_i] = \mathsf{TwoProd}(r_{i+1}, x)$
    $[r_i, \sigma_i] = \mathsf{TwoSum}(p_i, a_i)$
    $t = \pi_i \oplus \sigma_i$
    $c_i = c_{i+1} \otimes x \oplus t$
    $b_i = b_{i+1} \otimes |x| \oplus |t|$
  end
  $[\overline{r}, \delta] = \mathsf{TwoSum}(r_0, c_0)$
  if $|x| \leq 1$, $\rho = 1$
  else $\rho = \mathrm{fl}(|x|^{n-1}) \oslash (1+n\mathbf{u})$, end
  $\beta = |\delta| \oplus (\widehat{\gamma}_{4n+2} \otimes b_0 \oplus (2\mathbf{u} \otimes |\delta| \oplus (14n\mathbf{v} \otimes \rho \oplus 6\mathbf{v})))$

7.6 Conclusions

Dans ce chapitre, nous avons montré comment calculer une borne d'erreur *a posteriori* pour le résultat $\overline{r}$ calculé par le schéma de Horner compensé (algorithme 4.3). Ainsi, nous avons proposé une borne d'erreur plus fine que la borne *a priori* introduite chapitre 4. Pour établir cette borne d'erreur, nous avons pris en compte les erreurs d'arrondi commises lors du calcul du résultat compensé, ainsi que celles commises lors de l'évaluation de la borne elle-même. En l'absence d'underflow, nous sommes donc assurés d'obtenir une majoration de l'erreur directe directe $|\overline{r} - p(x)|$, et non pas une simple estimation de cette erreur. Nous avons également décrit une méthode permettant de tester *a posteriori* si l'évaluation compensée est un arrondi fidèle du résultat exact $p(x)$. L'algorithme CompHornerBound (algorithme 7.3) obtenu,

- calcule le résultat compensé $\overline{r} = \mathsf{CompHorner}(p, x)$ (algorithme 4.3),
- calcule d'une borne d'erreur *a posteriori* pour $\overline{r}$,
- teste si $\overline{r}$ est un arrondi fidèle de $p(x)$.

CompHornerBound est donc une version validée de CompHorner, puisqu'il permet d'obtenir des garanties sur la qualité de l'évaluation compensée.

L'algorithme CompHornerBound ne nécessite que des opérations élémentaires en arithmétique flottante, dans le mode d'arrondi au plus proche : nous n'utilisons ni bibliothèque pour le calcul d'intervalles, ni changement du mode d'arrondi. Tout comme CompHorner, cet algorithme est donc facilement portable, et ce montre très efficace en pratique : nos tests de performance montrent qu'il est environ deux fois moins coûteux de calculer une évaluation validée en précision doublée à l'aide de CompHornerBound, que de calculer un résultat de précision similaire, mais non validé, en arithmétique double-double. En outre, le temps d'exécution moyen observé pour CompHornerBound n'est qu'au plus 1.3 fois celui de CompHorner.

Chapitre 8

Schéma de Horner compensé $K-1$ fois

Plan du chapitre : À la Section 8.2, nous présentons la nouvelle transformation exacte EFTHornerK que nous utilisons pour décrire l'algorithme CompHornerK en Section 8.3. Nous démontrons dans la Section 8.3 que le résultat compensé produit par CompHornerK est aussi précis que s'il avait été calculé en K fois la précision de travail. Nous proposons à la Section 8.4 un algorithme permettant le calcul d'une borne d'erreur *a posteriori* pour le résultat compensé. La Section 8.5 est dédiée aux expériences numériques, visant à illustrer le comportement de CompHornerK, ainsi que la qualité des bornes obtenues. Les mesures de performances de la Section 8.7 montrent finalement l'intérêt pratique de ce nouvel algorithme compensé.

8.1 Introduction

Au chapitre 4, nous avons présenté le schéma de Horner compensé (algorithme 4.3). Le résultat calculé par cet algorithme est aussi précis que s'il avait été calculé par le schéma de Horner en précision doublée $\mathbf{u}^2$, avec un arrondi final vers la précision de travail $\mathbf{u}$. En particulier, la précision du résultat compensé est de l'ordre de l'unité d'arrondi tant que le conditionnement est petit devant $\mathbf{u}^{-1}$.

Dans le présent chapitre, nous généralisons le schéma de Horner compensé, et décrivons un nouvel algorithme compensé qui permet d'effectuer une évaluation polynomiale en K fois la précision de travail ($K \geq 2$). Le résultat compensé calculé par CompHornerK est en effet aussi précis que s'il était calculé par le schéma de Horner classique en précision $\mathbf{u}^K$, puis arrondi vers la précision $\mathbf{u}$. Nous démontrerons que l'erreur relative entachant l'évaluation compensée $\overline{r}$ satisfait une borne *a priori* de la forme

$$\frac{|\overline{r} - p(x)|}{|p(x)|} \leq \mathbf{u} + \mathcal{O}(\mathbf{u}^K)\,\mathrm{cond}(p, x),$$

où p est un polynôme à coefficients flottants et x un argument flottant.

Tout comme le schéma de Horner compensé du chapitre 4, l'algorithme d'évaluation CompHornerK repose sur une transformation exacte pour l'évaluation de $p(x)$. Le principe de cette nouvelle transformation est l'application récursive, sur $K-1$ niveaux, de la transformation exacte EFTHorner.

L'arrondi au plus proche est le mode d'arrondi courant, et sauf mention contraire, on supposera ici implicitement que les calculs flottants ne présentent aucun dépassement de capacité.

Rappelons le principe de la transformation exacte EFTHorner (algorithme 4.1, chapitre 4), en adoptant les notations qui seront utilisées dans ce chapitre. Soit p_1 un polynôme de degré n à coefficients flottants, et soit x un flottant. On considère

$$[h_1, p_2, p_3] = \mathsf{EFTHorner}(p_1, x),$$

avec h_1 un flottant, et p_1, p_2 deux polynômes à coefficients flottants de degré $n-1$. D'après le théorème 4.2, on a alors $h_1 = \mathsf{Horner}(p_1, x)$ et

$$p_1(x) = h_1 + (p_2 + p_3)(x).$$

Cette dernière égalité signifie précisément que EFTHorner est une transformation exacte pour l'évaluation de $p_1(x)$ par le schéma de Horner. En particulier, rappelons que, pour j variant de 0 à $n-1$,

- le coefficient de degré j de p_2 est l'erreur d'arrondi entachant le produit effectué à l'étape j de l'évaluation de $p_1(x)$ par le schéma de Horner ;
- le coefficient de degré j de p_3 est l'erreur d'arrondi générée par l'addition effectuée à l'étape j de cette même évaluation.

Dans le schéma de Horner compensé, les polynômes p_2 et p_3 sont directement utilisés afin de compenser les erreurs d'arrondi entachant $h_1 = \mathsf{Horner}(p_1, x)$, et le résultat compensé $\overline{r}$ est calculé de la manière suivante,

$$\overline{r} = h_1 \oplus \mathsf{Horner}(p_2 \oplus p_3, x)$$

Il est à noter que si le terme correctif $\mathsf{Horner}(p_2 \oplus p_3, x)$ est calculé exactement, et si $p(x)$ est un flottant, alors on a l'égalité $\overline{r} = p(x)$. Néanmoins, dans le cas général, le $p(x)$ exact n'est pas un flottant, et surtout le terme correctif n'est pas calculé exactement.

Ici nous proposons de ne plus utiliser directement les polynômes p_2 et p_3 pour compenser les erreurs d'arrondis entachant h_1, mais d'appliquer à nouveau EFTHorner pour évaluer ces polynômes : cela permet de compenser les erreurs d'arrondi générées en évaluant $p_2(x)$ et $p_3(x)$ en arithmétique flottante. On définit ainsi l'application récursive d'EFTHorner sur $K-1$ niveaux, et nous montrerons qu'il s'agit d'une nouvelle transformation exacte pour l'évaluation de $p_1(x)$, que nous appellerons EFTHornerK. Cette transformation exacte nous permettra de formuler l'algorithme CompHornerK, pour l'évaluation polynomiale en K fois la précision de travail.

Il est à noter qu'une précision de travail $\mathbf{u}^K$ peut également être simulée à l'aide de bibliothèques génériques. C'est le cas par exemple de la bibliothèque flottante multiprécision MPFR [31], qui permet de travailler en précision arbitraire[1], et donc en particulier en précision $\mathbf{u}^K$. Nous utiliserons cette bibliothèque comme référence pour illustrer les performances de l'algorithme CompHornerK.

Dans le cas d'un doublement de la précision de travail, nous avons déjà utilisé la bibliothèque double-double pour comparer ses performances à celle de l'algorithme CompHorner. La bibliothèque quad-double [35] permet de quadrupler la double précision de IEEE-754

[1] Cette arithmétique en précision arbitraire suit le modèle de l'arithmétique IEEE-754 [31]

et simule donc une précision de travail de l'ordre de 212 bits. Nous utiliserons cette bibliothèque pour évaluer les performances de l'algorithme CompHornerK dans le cas particulier où $K = 4$.

Ces comparaisons nous permettent de justifier l'intérêt pratique de l'algorithme CompHornerK : cet algorithme s'exécute en effet significativement plus rapidement que les alternatives basées sur les bibliothèques génériques indiquées ci-dessus, tant que $K \leq 4$. L'algorithme que nous proposons ici pourra donc être utilisé avec avantage pour doubler, tripler ou quadrupler la précision de travail.

8.2 Application récursive d'EFTHorner

On considère un polynôme p_1 de degré n à coefficients flottants, que l'on souhaite évaluer en un flottant x. Étant donné un entier $K \geq 2$, on définit dans cette section une nouvelle transformation exacte pour l'évaluation de $p_1(x)$ par le schéma de Horner, dont le principe est l'application récursive de l'algorithme EFTHorner (algorithme 4.1) sur $K - 1$ niveaux.

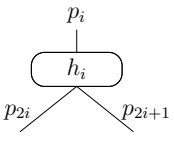
La transformation exacte EFTHorner nous sert donc de brique de base pour la conception d'une nouvelle transformation exacte. Nous décrivons dans un premier temps le principe de cette transformation sans erreur dans le cas particulier où $K = 3$, avant de passer au cas général.

8.2.1 Cas où $K = 3$

Commençons par décrire la manière dont nous représentons graphiquement une application de la transformation exacte EFTHorner (algorithme 4.1). Soit un polynôme p_i de degré d, à coefficients flottants, et soit x un flottant. On considère le flottant h_i et les deux polynômes p_{2i} et p_{2i+1} de degré au plus $d - 1$ tels que

$$[h_i, p_{2i}, p_{2i+1}] = \mathsf{EFTHorner}(p_i, x).$$

Rappelons que, d'après le théorème 4.2, $h_i = \mathsf{Horner}(p_i, x)$ et $p_i(x) = h_1 + (p_{2i} + p_{2i+1})(x)$. Pour représenter cette transformation exacte de $p_i(x)$, nous utiliserons par la suite le graphique ci-dessous.



Il s'agit ici d'appliquer la transformation EFTHorner de manière récursive sur 2 niveaux : ce processus est illustré sur la figure 8.1, sous la forme d'un arbre binaire. Notons en particulier que les feuilles de l'arbre binaire sont représentées par des rectangles, et portent les polynômes non évalués p_4, p_5, p_6 et p_7.

Définissons maintenant plus formellement ce processus de calcul, à l'aide de l'algorithme EFTHorner. On calcule :

- au niveau 1 : $[h_1, p_2, p_3] = \mathsf{EFTHorner}(p_1, x)$,
- au niveau 2 : $[h_2, p_4, p_5] = \mathsf{EFTHorner}(p_2, x)$ et $[h_3, p_6, p_7] = \mathsf{EFTHorner}(p_3, x)$.

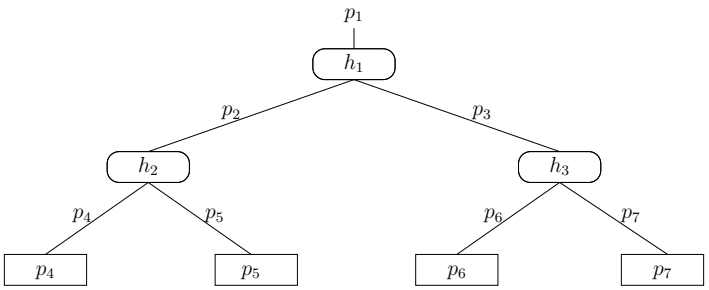


FIG. 8.1 – Illustration de la transformation exacte EFTHornerK dans le cas où $K = 3$.

Puisque $[h_1, p_2, p_3] = \mathsf{EFTHorner}(p_1, x)$, on a $p_1(x) = h_1 + (p_2 + p_3)(x)$. De plus, comme $[h_2, p_4, p_5] = \mathsf{EFTHorner}(p_2, x)$ et $[h_3, p_6, p_7] = \mathsf{EFTHorner}(p_3, x)$, on a respectivement $p_2(x) = h_2 + (p_4 + p_5)(x)$ et $p_3(x) = h_3 + (p_6 + p_7)(x)$. On obtient donc

$$p_1(x) = h_1 + h_2 + h_3 + (p_4 + p_5 + p_6 + p_7)(x), \tag{8.1}$$

où $h_1, h_2, h_3 \in \mathbf{F}$ et p_4, p_5, p_6, p_7 sont des polynômes à coefficients flottants de degré $n-2$.

L'équation (8.1) montre qu'en appliquant récursivement EFTHorner, on obtient naturellement une nouvelle transformation exacte pour l'évaluation de $p_1(x)$. Nous formulons ci-dessous cette transformation exacte.

Algorithme 8.1. Application récursive d'EFTHorner sur 2 niveaux.

fonction $[h_1, h_2, h_3, p_4, p_5, p_6, p_7] = \mathsf{EFTHorner3}(p_1, x)$
$[h_1, p_2, p_3] = \mathsf{EFTHorner}(p_1, x)$
$[h_2, p_4, p_5] = \mathsf{EFTHorner}(p_2, x)$
$[h_3, p_6, p_7] = \mathsf{EFTHorner}(p_3, x)$

Dans la sous-section suivante, nous généralisation cet algorithme, en appliquant récursivement EFTHorner sur $K \geq 2$ niveaux.

8.2.2 Cas général

On considère un polynôme p_1 à coefficients flottants de degré n, ainsi que $x \in \mathbf{F}$. Le principe de l'algorithme EFTHornerK est illustré sur la figure 8.2 sous la forme d'un arbre binaire à K niveaux. Notons bien que ce processus de calcul généralise simplement celui illustré sur la figure 8.1 (cas où $K = 3$). Aux niveaux 1 à $K-1$, on applique récursivement la transformation EFTHorner : on obtient ainsi 2^{K-1} polynômes au niveau K.

Rappelons que si on applique EFTHorner à un polynôme de degré d, alors les deux polynômes d'erreurs produits sont de degré $d-1$. Puisque p_1 est de degré n, et que l'on applique EFTHorner sur $K-1$ niveaux, cela signifie que les polynômes obtenus au niveau K sont de degré au plus $n-K+1$. En particulier, si $n-K+1 = 0$, les polynômes calculés aux feuilles de l'arbre sont de degré 0 : il est donc inutile de leur appliquer à nouveau EFTHorner. Pour simplifier la présentation, nous supposerons donc toujours que $n+1 \geq K \geq 2$.

Afin de référencer plus facilement les nœuds de l'arbre de la figure 8.2 définissons les ensembles d'indices suivants :

– $\mathcal{N}_T^K = \{1, \ldots, 2^K - 1\}$ est l'ensemble des indices des nœuds l'arbre,

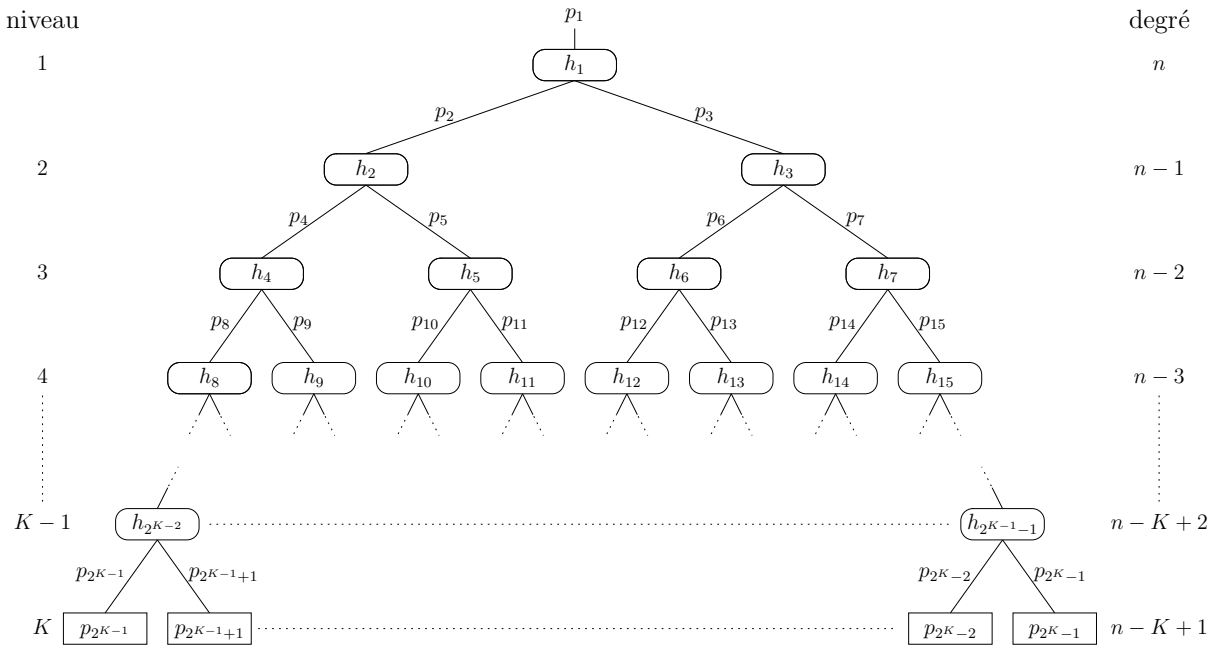


FIG. 8.2 – Illustration de la transformation exacte EFTHornerK

– $\mathcal{N}_I^K = \{1, \ldots, 2^{K-1}-1\}$ l'ensemble des indices des nœuds internes,
– $\mathcal{N}_L^K = \{2^{K-1}, \ldots, 2^K-1\}$ l'ensemble des indices feuilles de l'arbre.

Notons bien que

$$\mathcal{N}_T = \mathcal{N}_I \cup \mathcal{N}_L \quad \text{et} \quad \mathcal{N}_I \cap \mathcal{N}_L = \emptyset.$$

Précisons la cardinalité de ces ensembles, qui sera utile par la suite :

$$\text{card}(\mathcal{N}_T^K) = 2^K - 1, \quad \text{card}(\mathcal{N}_I^K) = 2^{K-1} - 1, \quad \text{et} \quad \text{card}(\mathcal{N}_L^K) = 2^{K-1}.$$

Sauf lorsqu'ils seront nécessaires, nous supprimerons les exposants K pour alléger les notations des ensembles $\mathcal{N}_T^K$, $\mathcal{N}_I^K$ et $\mathcal{N}_L^K$.

L'application récursive de la transformation exacte EFTHorner sur $K-1$ niveaux est alors définie par la relation :

$$[h_i, p_{2i}, p_{2i+1}] = \mathsf{EFTHorner}(p_i, x), \quad \text{pour} \quad i \in \mathcal{N}_I, \tag{8.2}$$

avec $h_i \in \mathbf{F}$ pour $i \in \mathcal{N}_I$ et p_i un polynôme à coefficients flottants pour chaque $i \in \mathcal{N}_T$. D'après le théorème 4.2, chacun des h_i ainsi défini est l'évaluation du polynôme p_i en x par le schéma de Horner, c'est à dire

$$h_i = \mathsf{Horner}(p_i, x), \quad \text{pour} \quad i \in \mathcal{N}_I. \tag{8.3}$$

D'autre part, comme EFTHorner est une transformation exacte pour le schéma de Horner, le théorème 4.2 montre que

$$p_i(x) = h_i + (p_{2i} + p_{2i+1})(x), \quad \text{pour} \quad i \in \mathcal{N}_I. \tag{8.4}$$

La relation (8.4) nous permettra de montrer dans la sous-section suivante que le processus de calcul que nous définissons ici constitue bien une transformation exacte pour l'évaluation de $p_1(x)$.

Les flottants $h_{i \in \mathcal{N}_I}$ et les polynômes $p_{i \in \mathcal{N}_L}$ sont calculés à l'aide de l'algorithme EFTHornerK suivant.

Algorithme 8.2. Application récursive d'EFTHorner sur $K-1$ niveaux.

fonction $[h_{i\in\mathcal{N}_I}, p_{i\in\mathcal{N}_L}] = \mathsf{EFTHornerK}(p_1, x)$
  pour $i \in \mathcal{N}_I$
    $[h_i, p_{2i}, p_{2i+1}] = \mathsf{EFTHorner}(p_i, x)$
  fin

Dans la sous-section suivante, nous montrons que l'algorithme 8.2 est une transformation exacte pour l'évaluation de $p_1(x)$, et nous en énonçons les propriétés numériques.

8.2.3 Propriétés numériques d'EFTHornerK

Commençons par montrer que l'algorithme EFTHornerK (algorithme 8.2) est bien une transformation exacte pour l'évaluation de $p_1(x)$.

Théorème 8.3. *Soit p_1 un polynôme de degré n à coefficients flottants, et soit x un flottant. Étant donné un entier K, avec $n+1 \geq K \geq 2$, on considère les flottants $h_{i\in\mathcal{N}_I}$, et les polynômes $p_{i\in\mathcal{N}_L}$, tels que $[h_{i\in\mathcal{N}_I}, p_{i\in\mathcal{N}_L}] = \mathsf{EFTHornerK}(p_1, x)$ (algorithme 8.2). Alors on a l'égalité suivante,*

$$p_1(x) = \sum_{i\in\mathcal{N}_I} h_i + \sum_{i\in\mathcal{N}_L} p_i(x). \tag{8.5}$$

L'algorithme EFTHornerK calcule l'évaluation $h_i = \mathsf{Horner}(p_i, x)$ de chacun des polynômes p_i, pour $i \in \mathcal{N}_I$. Pour la preuve du théorème 8.3, nous avons également besoin de considérer l'évaluation des $p_i(x)$, pour $i \in \mathcal{N}_L$. Nous définissons donc

$$h_i = \mathsf{Horner}(p_i, x), \quad \text{pour} \quad i \in \mathcal{N}_L.$$

Preuve du théorème 8.3. On procède par induction sur le paramètre K. Pour $K = 2$, on a bien $[h_1, p_2, p_3] = \mathsf{EFTHorner}(p_1, x)$. En effet, d'après le théorème 4.2, on a

$$p_1(x) = \mathsf{Horner}(p_1, x) + (p_2 + p_3)(x) = h_1 + p_2(x) + p_3(x).$$

Supposons maintenant que (8.5) soit satisfaite pour un certain K tel que $n+1 > K \geq 2$. On a ainsi

$$p_1(x) = \sum_{i\in\mathcal{N}_I^K} h_i + \sum_{i\in\mathcal{N}_L^K} p_i(x). \tag{8.6}$$

On considère alors les polynômes $p_{2^K}, \ldots, p_{2^{K+1}-1}$ définis par

$$[h_i, p_{2i}, p_{2i+1}] = \mathsf{EFTHorner}(p_i, x), \quad \text{pour} \quad i \in \mathcal{N}_L^K.$$

Pour $i \in \mathcal{N}_L^K$, d'après le théorème 4.2, on sait que $p_i(x) = h_i + (p_{2i} + p_{2i+1})(x)$. Donc

$$\sum_{i\in\mathcal{N}_L^K} p_i(x) = \sum_{i\in\mathcal{N}_L^K} h_i + \sum_{i\in\mathcal{N}_L^K} (p_{2i} + p_{2i+1})(x) = \sum_{i\in\mathcal{N}_L^K} h_i + \sum_{i\in\mathcal{N}_L^{K+1}} p_i(x).$$

En reportant cette dernière relation dans relation (8.6), on obtient

$$p_1(x) = \sum_{i\in\mathcal{N}_I^K} h_i + \sum_{i\in\mathcal{N}_L^K} h_i + \sum_{i\in\mathcal{N}_L^{K+1}} p_i(x) = \sum_{i\in\mathcal{N}_I^{K+1}} h_i + \sum_{i\in\mathcal{N}_L^{K+1}} p_i(x)$$

ce qui conclut la preuve du théorème 8.3. ■

Le fait qu'EFTHornerK constitue bien une transformation sans erreur nous permet alors de démontrer la proposition suivante.

Proposition 8.4. *Sous les hypothèses du théorème 8.3,*

$$p_1(x) - \sum_{i\in\mathcal{N}_T} h_i = \sum_{i\in\mathcal{N}_L} p_i(x) - h_i. \tag{8.7}$$

Preuve. Puisque l'algorithme EFTHornerK est une transformation exacte, en utilisant (8.5), on écrit

$$p_1(x) - \sum_{i\in\mathcal{N}_T} h_i = \sum_{i\in\mathcal{N}_I} h_i + \sum_{i\in\mathcal{N}_L} p_i(x) - \sum_{i\in\mathcal{N}_T} h_i.$$

Or, $\mathcal{N}_T = \mathcal{N}_I \cup \mathcal{N}_L$ et $\mathcal{N}_I \cap \mathcal{N}_L = \emptyset$, d'où

$$\begin{aligned} p_1(x) - \sum_{i\in\mathcal{N}_T} h_i &= \sum_{i\in\mathcal{N}_I} h_i + \sum_{i\in\mathcal{N}_L} p_i(x) - \sum_{i\in\mathcal{N}_I} h_i - \sum_{i\in\mathcal{N}_L} h_i \\ &= \sum_{i\in\mathcal{N}_L} p_i(x) - h_i, \end{aligned}$$

ce qui démontre la proposition 8.4. ■

L'égalité (8.7) signifie que l'erreur directe commise en approchant $p_1(x)$ par la somme exacte $\sum_{i\in\mathcal{N}_T} h_i$ est égale à la somme des erreurs commises en approchant chacun des $p_i(x)$, pour $i \in \mathcal{N}_L$, par $h_i = \mathsf{Horner}(p_i, x)$. La proposition suivante fournit en outre une majoration *a priori* de cette erreur directe, en faisant intervenir $\widetilde{p_1}(x) = \sum_{i=0} |a_i||x|^i$.

Proposition 8.5. *Sous les hypothèses du théorème 8.3, on a*

$$\left| p_1(x) - \sum_{i\in\mathcal{N}_T} h_i \right| \leq \gamma_{2(n-K+1)}\gamma_{4n}^{K-1}\widetilde{p_1}(x). \tag{8.8}$$

Preuve. D'après le proposition 8.4, en utilisant l'inégalité triangulaire,

$$\left| p_1(x) - \sum_{i\in\mathcal{N}_T} h_i \right| \leq \sum_{i\in\mathcal{N}_L} |p_i(x) - h_i|.$$

Les polynômes $p_{i\in\mathcal{N}_L}$ sont de degré $n-K+1$. Puisque $h_i = \mathsf{Horner}(p_i, x)$, on a donc

$$|p_i(x) - h_i| \leq \gamma_{2(n-K+1)}\widetilde{p_i}(x), \quad \text{pour} \quad i \in \mathcal{N}_L.$$

Puisque $[h_i, p_{2i}, p_{2i+1}] = \mathsf{EFTHorner}(p_i, x)$, d'après le théorème 4.2 on a $\widetilde{p_\pi}(x) + \widetilde{p_\sigma}(x) \leq \gamma_{2n}\widetilde{p}(x)$, d'où en particulier

$$\widetilde{p_{2i}}(x) \leq \gamma_{2n}\widetilde{p_i}(x) \quad \text{et} \quad \widetilde{p_{2i+1}}(x) \leq \gamma_{2n}\widetilde{p_i}(x).$$

On peut donc montrer par récurrence que

$$\widetilde{p_i}(x) \leq \gamma_{2n}^{K-1}\widetilde{p_1}(x), \quad \text{pour} \quad i \in \mathcal{N}_L,$$

d'où

$$|p_i(x) - h_i| \leq \gamma_{2(n-K+1)}\gamma_{2n}^{K-1}\widetilde{p_1}(x), \quad \text{pour} \quad i \in \mathcal{N}_L.$$

Comme $\mathrm{card}(\mathcal{N}_L) = 2^{K-1}$, il vient donc

$$\left| p_1(x) - \sum_{i \in \mathcal{N}_T} h_i \right| \leq \gamma_{2(n-K+1)} \gamma_{2n}^{K-1} 2^{K-1} \widetilde{p_1}(x).$$

Comme $\gamma_{2n}^{K-1} 2^{K-1} \leq \gamma_{4n}^{K-1}$, l'inégalité (8.8) s'ensuit. ■

Notre approche est motivée par l'inégalité (8.8). Cette inégalité montre que lorsque le paramètre K augmente d'une unité, la distance entre $p_1(x)$ et la somme exacte $\sum_{i \in \mathcal{N}_T} h_i$ est diminuée d'un facteur γ_{4n}. Dans la section suivante, nous montrons qu'en calculant cette somme avec une précision suffisante, on obtient un algorithme aussi précis que le schéma de Horner exécuté en K fois la précision courante.

8.3 Algorithme CompHornerK

Dans cette section, nous formulons l'algorithme CompHornerK (algorithme 8.6), puis nous effectuons une analyse d'erreur *a priori* de l'évaluation compensée calculée par cet algorithme.

8.3.1 Principe de l'algorithme

Comme dans la section précédente, on considère un polynôme p_1 de degré n à coefficients flottants, que l'on souhaite évaluer en un flottant x. Notre but est ici de proposer un schéma d'évaluation pour $p_1(x)$ qui soit d'une précision similaire à celle du schéma de Horner exécuté en K fois la précision de travail, c'est à dire à la précision $\mathbf{u}^K$. Rappelons que les flottants $h_{i \in \mathcal{N}_T}$ sont définis comme dans la section précédente par

$$[h_{i \in \mathcal{N}_I}, p_{i \in \mathcal{N}_L}] = \mathsf{EFTHornerK}(p_1, x),$$

et

$$h_i = \mathsf{Horner}(p_i, x), \quad \text{pour} \quad i \in \mathcal{N}_L.$$

Alors l'inégalité (8.8) montre que

$$\left| p_1(x) - \sum_{i \in \mathcal{N}_T} h_i \right| \leq (4n\mathbf{u})^K \widetilde{p_1}(x) + \mathcal{O}(\mathbf{u}^{K+1}). \tag{8.9}$$

Le principe de l'algorithme que nous proposons ci-dessous est de calculer une valeur approchée $\overline{r}$ de $\sum_{i \in \mathcal{N}_T} h_i$ en K fois la précision de travail, de manière à obtenir

$$|\overline{r} - p_1(x)| \leq \left(\mathbf{u} + \mathcal{O}(\mathbf{u}^2)\right) |p_1(x)| + \left((4n\mathbf{u})^K + \mathcal{O}(\mathbf{u}^{K+1})\right) \widetilde{p_1}(x),$$

ce qui donne en terme d'erreur relative :

$$\frac{|\overline{r} - p_1(x)|}{|p_1(x)|} \leq \left(\mathbf{u} + \mathcal{O}(\mathbf{u}^2)\right) + \left((4n\mathbf{u})^K + \mathcal{O}(\mathbf{u}^{K+1})\right) \mathrm{cond}(p_1, x). \tag{8.10}$$

Dans la borne précédente, le facteur multiplicatif de $\mathrm{cond}(p_1, x)$ en $(4n\mathbf{u})^K + \mathcal{O}(\mathbf{u}^{K+1})$ reflète une précision des calculs intermédiaires de l'ordre de $\mathbf{u}^K$. Le premier terme $\mathbf{u} + \mathcal{O}(\mathbf{u}^2)$ reflète quant à lui l'arrondi final du résultat vers la précision de travail $\mathbf{u}$.

Pour effectuer la sommation finale, nous utilisons l'algorithme SumK (algorithme 3.21) proposé dans [70], et qui a été décrit au chapitre 3. Nous obtenons donc l'algorithme CompHornerK ci-dessous.

Algorithme 8.6. Schéma d'évaluation polynomiale en K fois la précision de travail

function $\overline{r} = \mathsf{CompHornerK}(p_1, x, K)$
  $[h_{i\in\mathcal{N}_I}, p_{i\in\mathcal{N}_L}] = \mathsf{EFTHornerK}(p_1, x)$
  for $i \in \mathcal{N}_L$
    $h_i = \mathsf{Horner}(p_i, x)$
  end
  $\overline{r} = \mathsf{SumK}\,(h_{i\in\mathcal{N}_T}, K)$

Nous ne proposons pas ici de décompte précis du nombre d'opérations flottantes effectuées par l'algorithme CompHornerK. Une première raison à cela est que nous ne sommes pas parvenu à exprimer ce décompte sous une forme réellement informative. D'autre part, nous avons vu au chapitre 5 que le décompte du nombre d'opérations flottantes ne permet pas nécessairement d'expliquer les performances pratiques d'un algorithme. Dans ce chapitre, nous en resterons donc aux notations en $\mathcal{O}$ pour exprimer le coût, en nombre d'opérations flottantes, de nos algorithmes.

Dans l'algorithme EFTHornerK, les polynômes $p_{i\in\mathcal{N}_I}$ manipulés sont de degré au plus n : le traitement de chacun de ces polynômes par l'algorithme EFTHorner nécessite donc $\mathcal{O}(n)$ opérations flottantes. Comme $\mathrm{card}(\mathcal{N}_I) = 2^{K-1} - 1$, le coût de l'algorithme EFTHornerK est de $\mathcal{O}\big(n2^K\big)$ opérations flottantes. Dans CompHornerK, l'évaluation des 2^{K-1} polynômes $p_{i\in\mathcal{N}_L}$ nécessite également $\mathcal{O}(n2^K)$ opérations flottantes. Finalement, le calcul de $\mathsf{SumK}\,(h_{i\in\mathcal{N}_T}, K)$ (algorithme 3.21) nécessite $(6K-5)(2^{K-1}-1) = \mathcal{O}(n2^K)$ opérations flottantes.

On pourra donc retenir que le coût de CompHornerK est de $\mathcal{O}\big(n2^K\big)$ opérations flottantes. En particulier, à K fixé, notre algorithme nécessite $\mathcal{O}(n)$ opérations flottantes.

8.3.2 Borne d'erreur *a priori*

Le théorème suivant nous fournit une borne sur l'erreur directe entachant le résultat calculé par l'algorithme CompHornerK.

Théorème 8.7. *Soient p_1 un polynôme à coefficients flottants de degré n, x un flottant, et K tel que $n+1 \geq K \geq 2$. On suppose de plus que $(2^K-2)\gamma_{2n+1} \leq 1$. Alors l'erreur directe entachant le résultat $\overline{r} = \mathsf{CompHornerK}(p_1, x)$ de l'évaluation de $p_1(x)$ calculé à l'aide de l'algorithme 8.6 est majorée comme suit,*

$$|\overline{r} - p_1(x)| \leq \left(\mathbf{u} + 3\gamma^2_{2^K-2} + \gamma^K_{2^{K+1}-4}\right)|p_1(x)| + \left(\gamma^K_{4n} + \gamma_{2n+1}\gamma^K_{2^{K+1}-4} + \gamma^{K+1}_{4n}\right)\widetilde{p_1}(x). \quad (8.11)$$

Comme dans le cas du schéma de Horner compensé présenté au chapitre chapitre 4, il est important d'interpréter la borne d'erreur (8.11) en fonction du nombre de conditionnement $\mathrm{cond}(p_1, x)$, défini par l'égalité (2.9). On obtient ainsi la majoration (8.10) que nous rappelons ici :

$$\frac{|\overline{r} - p_1(x)|}{|p_1(x)|} \leq \left(\mathbf{u} + \mathcal{O}(\mathbf{u}^2)\right) + \left((4n\mathbf{u})^K + \mathcal{O}(\mathbf{u}^{K+1})\right)\mathrm{cond}(p_1, x). \quad (8.12)$$

Comme cela a déjà été dit en introduction de cette section, l'inégalité précédente signifie que le résultat compensé $\overline{r}$ est aussi précis que s'il avait été calculé par le schéma de Horner dans une précision de travail de l'ordre de $\mathbf{u}^K$, puis arrondi vers la précision courante $\mathbf{u}$.

Pour la preuve du théorème 8.7, nous utilisons le lemme suivant, qui fournit une majoration du conditionnement absolu de la somme des $h_{i\in\mathcal{N}_T}$.

Lemme 8.8. *On utilise ici les notations de l'algorithme 8.6. En supposant* $(2^K-2)\gamma_{2n+1} \leq 1$*, on a*

$$\sum_{i\in\mathcal{N}_T} |h_i| \leq |p_1(x)| + \gamma_{4n}\widetilde{p_1}(x). \tag{8.13}$$

Preuve. On décompose cette somme comme suit,

$$\sum_{i\in\mathcal{N}_T} |h_i| = |h_1| + \sum_{i\in\mathcal{N}_T-\{1\}} |h_i|.$$

Comme $h_1 = \mathsf{Horner}(p,x)$, $|h_1| \leq |p_1(x)| + \gamma_{2n}\widetilde{p}(x)$. D'autre part, pour $i \in \mathcal{N}_T - \{1\}$ on a également $h_i = \mathsf{Horner}(p_i,x)$, avec p_i de degré au plus $n-1$, donc

$$|h_i| \leq |p_i(x)| + \gamma_{2(n-1)}\widetilde{p_i}(x) \leq (1+\gamma_{2(n-1)})\widetilde{p_i}(x).$$

De plus, pour $i \in \mathcal{N}_T - \{1\}$ on sait que $\widetilde{p_i}(x) \leq \gamma_{2n}\widetilde{p}(x)$, d'où

$$|h_i| \leq (1+\gamma_{2(n-1)})\gamma_{2n}\widetilde{p}(x) \leq \gamma_{2n+1}\widetilde{p}(x).$$

Ainsi

$$\sum_{i\in\mathcal{N}_T} |h_i| \leq |p_1(x)| + \gamma_{2n}(1+(2^K-2)\gamma_{2n+1})\widetilde{p}(x).$$

Comme on a supposé $(2^K-2)\gamma_{2n+1} \leq 1$, on a

$$\gamma_{2n}(1+(2^K-2)\gamma_{2n+1}) \leq 2\gamma_{2n} \leq \gamma_{4n},$$

et l'inégalité (8.13) s'ensuit. ■

Preuve du théorème 8.7. On décompose l'erreur $|p_1(x)-\overline{r}|$ comme suit,

$$|\overline{r}-p_1(x)| \leq \underbrace{\left|p_1(x) - \sum_{i\in\mathcal{N}_T} h_i\right|}_{=e_1} + \underbrace{\left|\sum_{i\in\mathcal{N}_T} h_i - \overline{r}\right|}_{=e_2}. \tag{8.14}$$

Le premier terme e_1 dans (8.14) est majoré facilement à l'aide de la proposition 8.5. En effet,

$$e_1 \leq \gamma_{2(n-K+1)}\gamma_{4n}^{K-1}\widetilde{p}(x) \leq \gamma_{2n}\gamma_{4n}^{K-1}\widetilde{p}(x).$$

Le second terme e_2, désigne l'erreur commise lors de la sommation finale à l'aide de SumK. En utilisant la borne (3.14) sur l'erreur directe entachant le résultat calculé par SumK, on écrit

$$e_2 \leq (\mathbf{u}+3\gamma_{2^K-2}^2)\left|\sum_{i\in\mathcal{N}_T} h_i\right| + \gamma_{2^{K+1}-4}^K \sum_{i\in\mathcal{N}_T} |h_i|.$$

D'après le théorème 8.3,

$$\left|\sum_{i\in\mathcal{N}_T} h_i\right| \leq |p_1(x)| + \left|\sum_{i\in\mathcal{N}_L} h_i - p_i(x)\right| \leq |p_1(x)| + \gamma_{2n}\gamma_{4n}^{K-1}\widetilde{p}(x).$$

D'autre part, $\sum_{i\in\mathcal{N}_T} |h_i|$ est majorée à l'aide du lemme 8.8. D'où

$$\begin{aligned} e_2 &\leq (\mathbf{u} + 3\gamma_{2^K-2}^2)\left(|p_1(x)| + \gamma_{2n}\gamma_{4n}^{K-1}\widetilde{p}(x)\right) + \gamma_{2^{K+1}-4}^K\left(|p_1(x)| + \gamma_{4n}\widetilde{p}(x)\right) \\ &\leq \left(\mathbf{u} + 3\gamma_{2^K-2}^2 + \gamma_{2^{K+1}-4}^K\right)|p_1(x)| + \left(\gamma_{4n}\gamma_{2^{K+1}-4}^K + (\mathbf{u} + 3\gamma_{2^K-2}^2)\gamma_{2n}\gamma_{4n}^{K-1}\right)\widetilde{p}(x) \\ &\leq \left(\mathbf{u} + 3\gamma_{2^K-2}^2 + \gamma_{2^{K+1}-4}^K\right)|p_1(x)| + \left(\gamma_{4n}\gamma_{2^{K+1}-4}^K + \gamma_{4n}^{K+1}\right)\widetilde{p}(x). \end{aligned}$$

Ainsi

$$\begin{aligned} |\overline{r} - p_1(x)| &\leq \left(\mathbf{u} + 3\gamma_{2^K-2}^2 + \gamma_{2^{K+1}-4}^K\right)|p_1(x)| + \left(\gamma_{2n}\gamma_{4n}^{K-1} + \gamma_{4n}\gamma_{2^{K+1}-4}^K + \gamma_{4n}^{K+1}\right)\widetilde{p}(x) \\ &\leq \left(\mathbf{u} + 3\gamma_{2^K-2}^2 + \gamma_{2^{K+1}-4}^K\right)|p_1(x)| + \left(\gamma_{4n}^K + \gamma_{4n}\gamma_{2^{K+1}-4}^K + \gamma_{4n}^{K+1}\right)\widetilde{p}(x), \end{aligned}$$

ce qui achève la preuve du théorème 8.7. ■

8.4 Validation de CompHornerK

Le but de cette section est de proposer une version validée de CompHornerK, de la même manière qu'au chapitre 7 nous avons décrit comment valider le schéma de Horner compensé. Pour ce faire, il nous faut utiliser une version validée de l'algorithme SumK (algorithme 3.21) pour la sommation en K fois la précision de travail.

Nous rappelons donc ci-dessous la méthode utilisée dans [70] par Ogita, Rump et Oishi pour valider le résultat calculé par SumK, en y apportant une légère amélioration. Nous proposons ensuite une borne d'erreur *a posteriori* pour le résultat compensé calculé par CompHornerK.

8.4.1 Validation de SumK

La proposition suivante peut être utilisée afin de valider dynamiquement le résultat calculé par les algorithmes Sum2(algorithme 3.19) et SumK(algorithme 3.21). Rappelons que $\overline{s} = p_n \oplus c$ désigne le résultat compensé calculé par l'un ou l'autre de ces algorithmes.

Proposition 8.9 (corollaire 4.7 dans [70]). *Soit un vecteur de flottants $p = (p_1, \ldots, p_n)$ et soit $s = \sum_{i=1}^n p_i$. Si on ajoute les lignes*

```
if 2nu ≤ 1
   error('echec de la validation')
end
β = γ̂_{2n} ⊗ Sum(|p_1|, ..., |p_n|)
μ = u|s̄| ⊕ (β ⊕ (2u²|s̄| ⊕ 3v))
```

à l'algorithme 3.19 ou à algorithme 3.21, alors dans les deux cas, même en présence d'underflow,

$$|s - \overline{s}| \leq \mu.$$

Nous proposons ci-dessous une amélioration de la borne d'erreur dynamique de la proposition 8.9. Cette amélioration est basée sur l'observation suivante : dans la proposition 8.9, l'erreur d'arrondi δ entachant l'addition finale $\overline{s} = p_n \oplus c$, effectuée par les algorithmes Sum2 et SumK, est majorée par $\mathbf{u}|\overline{s}|$. Or, en pratique, cette erreur d'arrondi peut être négligeable devant les autres sources d'erreurs prises en compte dans le calcul de $\mu = \mathbf{u}|\overline{s}| \oplus (\beta \oplus (2\mathbf{u}^2|\overline{s}| \oplus 3\mathbf{v}))$. Nous proposons donc de ne plus simplement majorer l'erreur d'arrondi δ, mais de la calculer exactement, à l'aide de la transformation exacte TwoSum (algorithme 3.5). Nous formulons ci-dessous l'algorithme SumKBound, qui calcule à la fois le résultat compensé $\overline{s}$ et une borne sur l'erreur directe entachant ce résultat.

Algorithme 8.10. Sommation compensée K fois.

function $[\overline{s}, \nu] = \mathsf{SumKBound}(p, K)$
  if $2n\mathbf{u} \leq 1$
    error('echec de la validation')
  end
  for $k = 1 : K-1$
    $p = \mathsf{VecSum}(p)$
  end
  $c = \mathsf{Sum}\big(p_1, \ldots, p_{n-1}\big)$
  $[\overline{s}, \delta] = \mathsf{TwoSum}(p_n, c)$
  $\beta = \widehat{\gamma}_{2n} \otimes \mathsf{Sum}\big(|p_1|, \ldots, |p_n|\big)$
  $\nu = |\delta| \oplus (\beta \oplus (2\mathbf{u}|\delta| \oplus 2\mathbf{v}))$

Proposition 8.11. *Soit un vecteur de flottants $p = (p_1, \ldots, p_n)$ et soit $s = \sum_{i=1}^{n} p_i$. Soient $\overline{s}$ et ν les flottants tels que $[\overline{s}, \nu] = \mathsf{SumKBound}(p, K)$. Alors, même en présence d'underflow,*

$$|s - \overline{s}| \leq \nu.$$

Preuve. Il suffit d'adapter la preuve de la proposition 8.9 proposée dans [70]. On note $p' = (p'_1, \ldots, p'_n)$ le vecteur obtenu après les $K-1$ itérations de la boucle pour. Comme $\overline{s} = p_n \oplus c$, en utilisant l'inégalité triangulaire,

$$|s - \overline{s}| = |p_n \oplus c - s| \leq \left|p_n \oplus c - \left(p'_n + c\right)\right| + |p'_n + c - s|\,.$$

Puisque $[\overline{s}, \delta] = \mathsf{TwoSum}(p'_n, c)$, le premier terme dans borne ci-dessus est exactement égal à $|\delta|$,

$$|s - \overline{s}| \leq |\delta| + |p'_n + c - s|.$$

Comme VecSum et une transformation exacte pour la sommation, on a $s = \sum_{i=1}^{n} p'_i$. D'autre part, rappelons que $c = \mathsf{Sum}\left(p'_1, \ldots, p'_{n-1}\right)$. On écrit donc

$$\begin{aligned} |s - \overline{s}| &\leq |\delta| + \left| c - \sum_{i=1}^{n-1} p'_i \right| \\ &\leq |\delta| + \gamma_{n-2} \sum_{i=1}^{n-1} |p'_i| \end{aligned}$$

On note $\alpha = \mathsf{Sum}\left(|p'_1|, \ldots, |p'_n|\right)$, ce qui permet d'écrire

$$\begin{aligned} |s - \overline{s}| &\leq |\delta| + (1+\mathbf{u})^{n-2}\gamma_{n-2}\alpha \\ &\leq |\delta| + \gamma_{2n-4}\alpha. \end{aligned}$$

Comme $\gamma_k \leq (1-\mathbf{u})\gamma_{k+1}$, il vient

$$\begin{aligned} |s-\overline{s}| &\leq |\delta| + (1-\mathbf{u})^4\gamma_{2n}\alpha \\ &\leq (1-\mathbf{u})|\delta| + (1-\mathbf{u})^2\beta + \mathbf{u}|\delta| \\ &\leq (1-\mathbf{u})|\delta| + (1-\mathbf{u})^2\beta + \mathbf{u}\otimes|\delta| + \mathbf{v}. \end{aligned}$$

Comme $1 \leq 2(1-\mathbf{u})^3$, on écrit,

$$\begin{aligned} |s-\overline{s}| &\leq (1-\mathbf{u})|\delta| + (1-\mathbf{u})^2\beta + (1-\mathbf{u})^3(2\mathbf{u}|\delta| + 2\mathbf{v}) \\ &\leq (1-\mathbf{u})|\delta| + (1-\mathbf{u})^2\beta + (1-\mathbf{u})^2(2\mathbf{u}|\delta| \oplus 2\mathbf{v}) \\ &\leq (1-\mathbf{u})|\delta| + (1-\mathbf{u})(\beta \oplus (2\mathbf{u}|\delta| \oplus 2\mathbf{v})) \\ &\leq |\delta| \oplus (\beta \oplus (2\mathbf{u}|\delta| \oplus 2\mathbf{v})) = \nu. \end{aligned}$$

Ceci achève la preuve de la proposition 8.11. ■

8.4.2 Borne d'erreur *a posteriori* pour CompHornerK

Nous proposons ci-dessous un algorithme calculant à la fois $\overline{r} = \mathsf{CompHornerK}(p, x, K)$ (algorithme 8.6) et une borne d'erreur *a posteriori* μ. Rappelons que l'algorithme Sum (algorithme 3.17) désigne la sommation récursive classique.

Algorithme 8.12. CompHornerK avec borne d'erreur validée

function $[\overline{r}, \mu] = \mathsf{CompHornerKBound}(p_1, x, K)$
  $d = n - K + 1$
  $k = 2d + 2^{K-1} + 3$
  if $k\mathbf{u} \geq 1$ then
    error('validation impossible')
  end
  $[h_{i\in\mathcal{N}_I}, p_{i\in\mathcal{N}_L}] = \mathsf{EFTHornerK}(p_1, x)$
  for $i \in \mathcal{N}_L$
    $h_i = \mathsf{Horner}(p_i, x)$
    $\widetilde{h}_i = \mathsf{Horner}(|p_i|, |x|)$
  end
  $\alpha = \big(\widehat{\gamma}_{2d} \otimes \mathsf{Sum}\big(\widetilde{h}_{i\in\mathcal{N}_L}\big)\big) \oslash (1-k\mathbf{u})$
  $[\overline{r}, \beta] = \mathsf{SumKBound}\big(h_{i\in\mathcal{N}_T}, K\big)$
  $\mu = (\alpha+\beta) \oslash (1-2\mathbf{u})$

Nous utilisons ici l'algorithme SumKBound (algorithme 8.10), afin de calculer une borne β sur l'erreur $\left|\sum_{i\in\mathcal{N}_T} h_i - \overline{r}\right|$ entachant la somme finale des $h_{i\in\mathcal{N}_T}$. On supposera toujours par la suite que la condition $k\mathbf{u} < 1$ est satisfaite, ce qui revient à supposer que l'exécution de l'algorithme se termine sans que l'erreur 'validation impossible' ne soit déclenchée. Le théorème suivant montre que la borne d'erreur μ calculée en arithmétique flottante est dans ce cas un majorant de l'erreur directe $|\overline{r} - p_1(x)|$.

Théorème 8.13. *On considère un polynôme p à coefficients flottants de degré n, x un flottant, et K un entier tel que $n+1 \geq K \geq 2$. Soient $\overline{r}$ et μ les flottants tels que $[\overline{r}, \mu] = \mathsf{CompHornerKBound}(p, x, K)$ (algorithme 8.12). Alors on a $\overline{r} = \mathsf{CompHornerK}(p, x, K)$ (algorithme 8.6) et, en l'absence d'underflow,*

$$|\overline{r} - p_1(x)| \leq \mu.$$

Preuve. Comme dans la preuve du théorème 8.7, on décompose l'erreur $|\overline{r} - p_1(x)|$ comme suit,

$$|\overline{r} - p_1(x)| \leq \underbrace{\left| p_1(x) - \sum_{i \in \mathcal{N}_T} h_i \right|}_{=e_1} + \underbrace{\left| \sum_{i \in \mathcal{N}_T} h_i - \overline{r} \right|}_{=e_2}. \tag{8.15}$$

Commençons par montrer que e_1 est majoré par α. On a, d'après la proposition 8.4,

$$e_1 = \left| \sum_{i \in \mathcal{N}_L} p_i(x) - h_i \right| \leq \sum_{i=2^{K-1}}^{2^K-1} |p_i(x) - h_i|.$$

Pour $i \in \mathcal{N}_L$, chacun des polynômes p_i, est de degré au plus $d = n - K + 1$. Comme $h_i = \mathsf{Horner}(p_i, x)$ et $\widetilde{h}_i = \mathsf{Horner}(|p_i|, |x|)$, on en déduit que

$$|h_i - p_i(x)| \leq \gamma_{2d} \widetilde{p}_i(x) \leq (1+\mathbf{u})^{2d} \gamma_{2d} \widetilde{h}_i,$$

pour $i \in \mathcal{N}_L$. D'où

$$e_1 \leq (1+\mathbf{u})^{2d} \gamma_{2d} \sum_{i \in \mathcal{N}_L} \widetilde{h}_i.$$

Notons ici $S := \mathsf{Sum}\big(\widetilde{h}_{i \in \mathcal{N}_L}\big)$. Comme S est une somme de 2^{K-1} termes positifs, calculée par l'algorithme de sommation récursive classique,

$$\sum_{i \in \mathcal{N}_L} \widetilde{h}_i \leq (1+\mathbf{u})^{2^{K-1}} S.$$

D'où,

$$\begin{aligned} e_1 &\leq (1+\mathbf{u})^{2d+2^{K-1}} \gamma_{2d} S \\ &\leq (1+\mathbf{u})^{2d+2^{K-1}+2} \widehat{\gamma}_{2d} \otimes S \\ &\leq (\widehat{\gamma}_{2d} \otimes S) \oslash \big(1 - (2d + 2^{K-1} + 3)\mathbf{u}\big) = \alpha. \end{aligned}$$

L'erreur e_1 est donc effectivement majorée par α. D'autre part, puisque que e_2 est l'erreur directe commise en approchant la somme finale $\sum_{i \in \mathcal{N}_T} h_i$ par $\overline{r} = \mathsf{SumK}(h_{i \in \mathcal{N}_L})$, d'après la proposition 8.11 on a $e_2 \leq \beta$. On obtient donc,

$$|\overline{r} - p_1(x)| \leq \alpha + \beta \leq (\alpha + \beta) \oslash (1 - 2\mathbf{u}),$$

ce qui conclut la preuve du théorème. ■

8.5 Expériences numériques

Toutes ces expériences sont réalisées en utilisant la double précision IEEE-754 comme précision de travail.

8.5.1 Précision du résultat

La figure 8.3 met en évidence le comportement numérique de l'algorithme CompHornerK (algorithme 8.6).

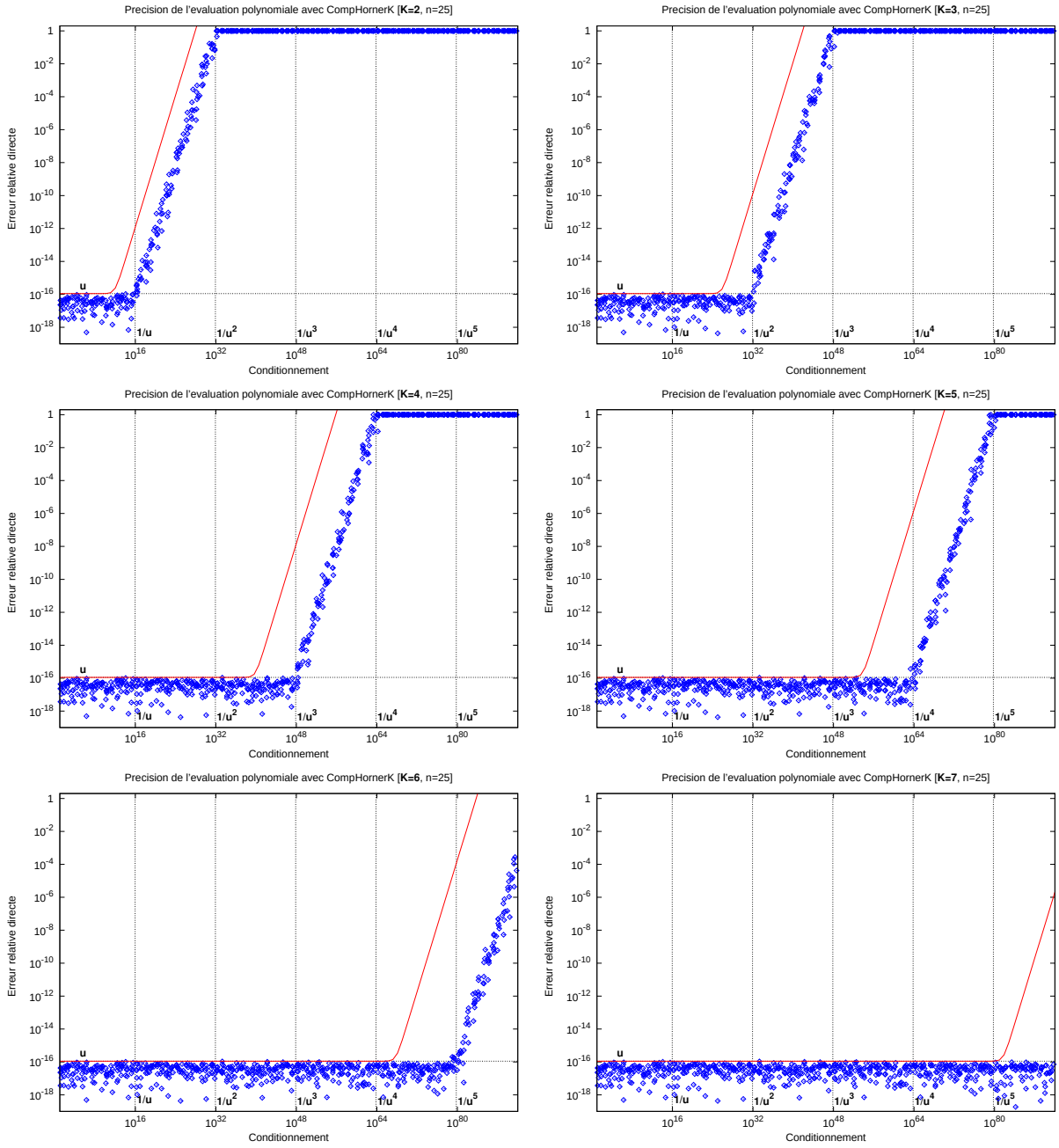


FIG. 8.3 – Précision du résultat calculé par CompHornerK (algorithme 8.6) en fonction du conditionnement $\mathrm{cond}(p, x)$, pour $K = 1, 2, \ldots, 7$.

Comme au chapitre 4 nous avons généré un jeu de 700 polynômes de degré 25, dont le conditionnement varie entre 10^2 et 10^{100}. Nous reportons sur la figure 8.3 la précision relative de chaque évaluation effectuée en fonction du nombre de conditionnement $\mathrm{cond}(p, x)$ de celle-ci, pour $K = 2, \ldots, 7$. Nous représentons également sur chaque graphique la borne

a priori (8.12) sur l'erreur relative entachant le résultat calculé.

On constate que l'on observe bien le comportement attendu : pour chaque valeur de K considérée, tant que le conditionnement est inférieur à $\mathbf{u}^{-K}$, la précision du résultat calculé est de l'ordre de l'unité d'arrondi $\mathbf{u}$. Cela signifie que l'algorithme CompHornerK est bien d'une précision similaire à celle du schéma de Horner effectué en précision $\mathbf{u}^K$, avec un arrondi final vers la précision courante $\mathbf{u}$. On notera également que pour $K=2$ on retrouve naturellement le même comportement que celui observé au chapitre 4 pour l'algorithme CompHorner.

On observe que la borne *a priori* (8.12) sur l'erreur relative du résultat calculé est toujours pessimiste de plusieurs ordres de grandeurs. De plus, cette borne est de plus en plus pessimiste au fur et à mesure que le paramètre K augmente. Ce phénomène est également observé dans [70] où est proposé un produit scalaire en K fois la précision de travail. Les expériences numériques qui y sont décrites montrent que la borne d'erreur *a priori* obtenue pour ce produit scalaire est également de plus en plus pessimiste lorsque K augmente.

8.5.2 Borne d'erreur *a posteriori*

Nos tests visent à mettre en évidence la finesse de la borne *a posteriori* obtenue à l'aide de l'algorithme CompHornerKBound (algorithme 8.12). Pour $K=3,\ldots,6$, nous évaluons avec l'algorithme CompHornerKBound le polynôme $p_{15}=(1-x)^{15}$ en 512 points au voisinage de la racine $x=1$. Les résultats de ce test sont reportés sur la figure 8.4.

En notant β la borne d'erreur *a priori* (8.11), et μ la borne d'erreur *a posteriori* calculée par CompHornerKBound, on reporte sur les graphiques de la figure 8.4 les ratios

$$\frac{\beta}{|p_1(x)|}, \quad \text{et} \quad \frac{\mu}{|p_1(x)|}.$$

Ces ratios représentent l'erreur relative maximale garantie respectivement par la borne *a priori* β et la borne *a posteriori* μ. Nous reportons également sur les graphiques l'erreur relative directe entachant le résultat $\overline{r}$ calculé par CompHornerKBound.

On observe, comme dans les expériences de la sous-section précédente, que plus on augmente la précision du calcul, plus la qualité du résultat calculé s'améliore. Naturellement, les bornes d'erreur, aussi bien celle *a priori* qu'*a posteriori*, suivent également cette évolution : lorsque l'on augmente le paramètre K, les bornes d'erreurs garantissent une erreur relative de plus en plus petite.

On observe également que la borne à *a posteriori* est toujours significativement plus fine que la borne *a priori*. Notamment, lorsque la précision de calcul est suffisante pour obtenir une erreur relative plus petite que l'unité d'arrondi $\mathbf{u}\approx 10^{-16}$ (voir sur les graphiques pour $K=4,5,6$), on constate que la borne *a posteriori* se confond avec l'erreur directe mesurée. Rappelons que cela a déjà été observé dans les expériences numériques du chapitre 7 a propos de la borne d'erreur fournie par l'algorithme CompHornerBound (algorithme 7.3).

8.6 Implantation pratique de CompHornerK

Tel que formulé à la Section 8.3, l'algorithme CompHornerK (algorithme 8.6) nécessite de calculer successivement, et de stocker en mémoire, les polynômes $p_{i\in\mathcal{N}_T}$. Ces polynômes

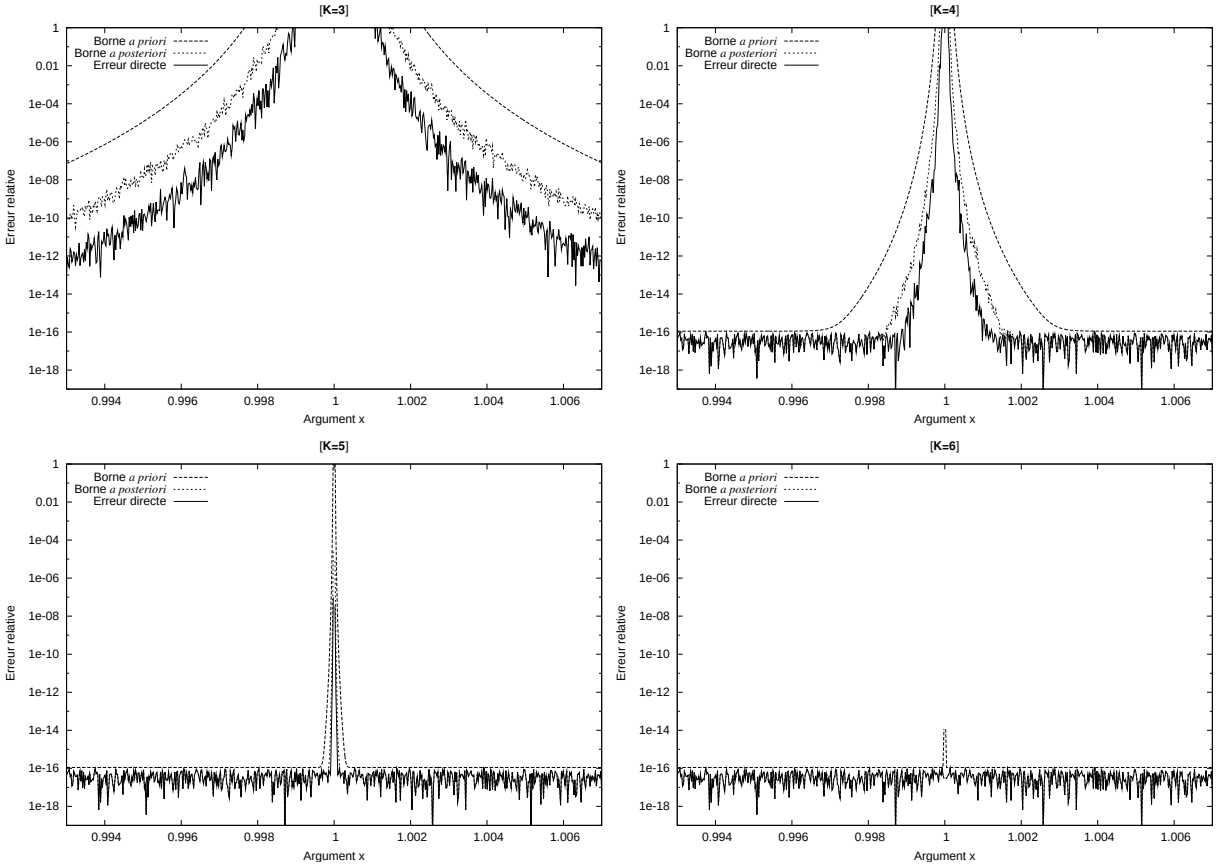


FIG. 8.4 – Comparaison de la borne d'erreur *a priori* (8.11) et de la borne *a posteriori* de l'algorithme CompHornerKBound (algorithme 8.12).

sont au nombre de $2^K - 1$, et leur degré est compris entre $n - K + 1$ et n : cela représente un espace de stockage intermédiaire de $\mathcal{O}(n2^K)$ flottants.

Afin d'obtenir un algorithme performant, il est important de diminuer la taille de cette espace de stockage intermédiaire, de manière à réduire les coûts associés en pratique aux accès à la mémoire. Dans cette section, nous montrons comment réduire l'espace de stockage nécessaire à l'implantation de CompHornerK, sans changer la complexité de l'algorithme, en terme de nombre d'opérations flottantes. Nous montrons en effet qu'un espace de stockage de $\mathcal{O}(2^K)$ flottants est suffisant.

Nous commencerons par étudier en détail le cas où $K = 3$, pour décrire ensuite plus succinctement le cas général.

8.6.1 Cas où $K = 3$

On considère un polynôme p_1 de degré n, à coefficients flottants, ainsi qu'une valeur x flottante. En fixant $K = 3$ dans l'algorithme 8.6, on obtient l'algorithme suivant.

Algorithme 8.14. CompHornerK dans le cas où $K = 3$

fonction $\overline{r} = \mathsf{CompHorner3}(p_1, x)$
  $[h_1, h_2, h_3, p_4, p_5, p_6, p_7] = \mathsf{EFTHorner3}(p_1, x)$

$h_4 = \mathsf{Horner}(p_4, x)$
$h_5 = \mathsf{Horner}(p_5, x)$
$h_6 = \mathsf{Horner}(p_6, x)$
$h_7 = \mathsf{Horner}(p_7, x)$
$\overline{r} = \mathsf{SumK}(h_1, \ldots, h_7, 3)$

Commençons par rappeler que le calcul de la transformation exacte $\mathsf{EFTHorner3}(p_1, x)$ s'effectue de la manière suivante :

$[h_1, p_2, p_3] = \mathsf{EFTHorner}(p_1, x)$
$[h_2, p_4, p_5] = \mathsf{EFTHorner}(p_2, x)$
$[h_3, p_6, p_7] = \mathsf{EFTHorner}(p_3, x)$

On désigne ici par $p_{i,j}$ le coefficient de degré j du polynôme p_i. Puisque $[h_1, p_2, p_3] = \mathsf{EFTHorner}(p_1, x)$, par définition d'EFTHorner (algorithme 4.1), rappelons que

- $p_{2,j}$ est l'erreur d'arrondi générée par le produit effectué à l'étape j de l'évaluation de $p_1(x)$ par le schéma de Horner,
- $p_{3,j}$ est l'erreur d'arrondi générée par l'addition effectuée à l'étape j de l'évaluation de $p_1(x)$ par le schéma de Horner.

Il en va de même pour les transformations $[h_2, p_4, p_5] = \mathsf{EFTHorner}(p_2, x)$ et $[h_3, p_6, p_7] = \mathsf{EFTHorner}(p_3, x)$. Le calcul de ces trois transformations exactes peut donc être écrit comme suit.

$h_1 = p_{1,n}$
for $j = n-1 : -1 : 0$
  $[h_1, p_{2,j}] = \mathsf{TwoProd}(h_1, x)$; $[h_1, p_{3,j}] = \mathsf{TwoProd}(h_1, p_{1,j})$
end
$h_2 = p_{2,n-1}$
for $j = n-2 : -1 : 0$
  $[h_2, p_{4,j}] = \mathsf{TwoProd}(h_2, x)$; $[h_2, p_{5,j}] = \mathsf{TwoProd}(h_2, p_{2,j})$
end
$h_3 = p_{3,n-1}$
for $j = n-2 : -1 : 0$
  $[h_3, p_{6,j}] = \mathsf{TwoProd}(h_3, x)$; $[h_3, p_{7,j}] = \mathsf{TwoProd}(h_3, p_{3,j})$
end

Il est aisé de voir que les trois boucles de la portion de code précédente peuvent être regroupées en une seule boucle. Il est à noter que les polynômes p_2 et p_3 sont de degré $n-1$, alors que p_1 est de degré n : il faut en tenir compte en insérant le prologue adéquat. Cela nous permet de ré-écrire l'algorithme 8.1 comme indiqué ci-dessous.

Algorithme 8.15. Application récursive d'EFTHorner sur 2 niveaux.

fonction $[h_1, h_2, h_3, p_4, p_5, p_6, p_7] = \mathsf{EFTHorner3}(p_1, x)$
  $h_1 = p_{1,n}$
  $[h_1, p_{2,n-1}] = \mathsf{TwoProd}(h_1, x)$; $[h_1, p_{3,n-1}] = \mathsf{TwoProd}(h_1, p_{1,n-1})$
  $h_2 = p_{2,n-1}$; $h_3 = p_{3,n-1}$
  for $j = n-2 : -1 : 0$
    $[h_1, p_{2,j}] = \mathsf{TwoProd}(h_1, x)$; $[h_1, p_{3,j}] = \mathsf{TwoProd}(h_1, p_{1,j})$

$[h_2, p_{4,j}] = \mathsf{TwoProd}(h_2, x)$; $[h_2, p_{5,j}] = \mathsf{TwoProd}(h_2, p_{2,j})$
$[h_3, p_{6,j}] = \mathsf{TwoProd}(h_3, x)$; $[h_3, p_{7,j}] = \mathsf{TwoProd}(h_3, p_{3,j})$
end

Nous combinons maintenant cette écriture de EFTHorner3 avec l'algorithme 8.14 pour obtenir une version de l'algorithme CompHorner3 ne présentant plus qu'une seule boucle : il suffit pour cela d'intégrer l'évaluation des polynômes p_4, p_5, p_6 et p_7 à la boucle principale. Puisque ces polynômes sont de degré $n-2$, cela nous oblige à dérouler à nouveau une itération de la boucle dans son prologue.

Algorithme 8.16. CompHornerK dans le cas où $K = 3$

fonction $\overline{r} = \mathsf{CompHorner3}(p_1, x)$
$h_1 = p_{1,n}$
$[h_1, p_{2,n-1}] = \mathsf{TwoProd}(h_1, x)$; $[h_1, p_{3,n-1}] = \mathsf{TwoProd}(h_1, p_{1,n-1})$
$h_2 = p_{2,n-1}$; $h_3 = p_{3,n-1}$
$[h_1, p_{2,n-2}] = \mathsf{TwoProd}(h_1, x)$; $[h_1, p_{3,n-2}] = \mathsf{TwoProd}(h_1, p_{1,n-2})$
$[h_2, p_{4,n-2}] = \mathsf{TwoProd}(p_{2,n-1}, x)$; $[h_2, p_{5,n-2}] = \mathsf{TwoProd}(h_2, p_{2,n-2})$
$[h_3, p_{6,n-2}] = \mathsf{TwoProd}(p_{3,n-1}, x)$; $[h_3, p_{7,n-2}] = \mathsf{TwoProd}(h_3, p_{3,n-2})$
$h_4 = p_{4,n-2}$; $h_5 = p_{5,n-2}$; $h_6 = p_{6,n-2}$; $h_7 = p_{7,n-2}$
for $j = n-3 : -1 : 0$
$[h_1, p_{2,j}] = \mathsf{TwoProd}(h_1, x)$; $[h_1, p_{3,j}] = \mathsf{TwoProd}(h_1, p_{1,j})$
$[h_2, p_{4,j}] = \mathsf{TwoProd}(h_2, x)$; $[h_2, p_{5,j}] = \mathsf{TwoProd}(h_2, p_{2,j})$
$[h_3, p_{6,j}] = \mathsf{TwoProd}(h_3, x)$; $[h_3, p_{7,j}] = \mathsf{TwoProd}(h_3, p_{2,j})$
$h_4 = h_4 \otimes x \oplus p_{4,j}$;
$h_5 = h_5 \otimes x \oplus p_{5,j}$
$h_6 = h_6 \otimes x \oplus p_{6,j}$;
$h_7 = h_7 \otimes x \oplus p_{7,j}$
end
$\overline{r} = \mathsf{SumK}(h_1, \ldots, h_7, 3)$

En pratique, les variables $h_1, \ldots, h_7$ pourront clairement être regroupées au sein d'un vecteur $h = (h_1, \ldots, h_7)$ de longueur 7. De plus, on voit que dans la boucle de l'algorithme 8.16, à l'itération i seuls les coefficients $p_{1,j}, \ldots, p_{7,j}$ de degré j interviennent. Ces coefficients pourront donc également être stockés et manipulés dans un vecteur $e = (e_1, \ldots, e_7)$. L'espace de stockage intermédiaire nécessaire est donc de 14 flottants. En prenant en compte ces remarques, et en supprimant les affectations inutiles, on obtient finalement l'algorithme ci-dessous, qui montre comment CompHorner3 peut être implanté efficacement.

Algorithme 8.17. CompHornerK dans le cas où $K = 3$

fonction $\overline{r} = \mathsf{CompHorner3}(p_1, x)$
$e_1 = p_{1,n}$
$[h_1, e_2] = \mathsf{TwoProd}(h_1, x)$; $[h_1, e_3] = \mathsf{TwoProd}(h_1, e_1)$
$[h_1, e_2] = \mathsf{TwoProd}(h_1, x)$; $[h_1, e_3] = \mathsf{TwoProd}(h_1, e_1)$
$[h_2, h_4] = \mathsf{TwoProd}(h_2, x)$; $[h_2, h_5] = \mathsf{TwoProd}(h_2, e_2)$
$[h_3, h_6] = \mathsf{TwoProd}(h_3, x)$; $[h_3, h_7] = \mathsf{TwoProd}(h_3, e_3)$
for $j = n-3 : -1 : 0$

$[h_1, e_2] = \mathsf{TwoProd}(h_1, x)$; $[h_1, e_3] = \mathsf{TwoProd}(h_1, e_1)$
$[h_2, e_4] = \mathsf{TwoProd}(h_2, x)$; $[h_2, e_5] = \mathsf{TwoProd}(h_2, e_2)$
$[h_3, e_6] = \mathsf{TwoProd}(h_3, x)$; $[h_3, e_7] = \mathsf{TwoProd}(h_3, e_2)$
$h_4 = h_4 \otimes x \oplus e_4$;
$h_5 = h_5 \otimes x \oplus e_5$
$h_6 = h_6 \otimes x \oplus e_6$;
$h_7 = h_7 \otimes x \oplus e_7$
end
$\overline{r} = \mathsf{SumK}(h, 3)$

8.6.2 Cas général

Nous présentons ici plus succinctement la manière dont nous avons implanté l'algorithme CompHornerK (algorithme 8.6). La méthode utilisée est exactement celle décrite ci-dessus dans le cas $K = 3$. L'écriture du prologue de la boucle principale est néanmoins rendue plus complexe du fait que K soit ici une entrée de l'algorithme. Comme dans l'algorithme 8.14, on utilise deux vecteurs de flottants $h = (h_1, \ldots, h_{2^K-1})$ et $e = (e_1, \ldots, e_{2^K-1})$ de longueurs $2^K - 1$:

- l'élément h_i du vecteur h sert de variable intermédiaire pour l'évaluation du polynôme p_i par le schéma de Horner,
- l'élément e_i du vecteur e permet stocker temporairement les coefficients du polynôme p_i.

L'espace de stockage intermédiaire nécessaire à l'implantation de CompHornerK se trouve donc réduit à la taille de $\mathcal{O}(2^K)$ flottants. Ceci nous donne l'algorithme algorithme 8.18 ci-dessous. Nous conservons le nom CompHornerK pour cet algorithme, puisque seul l'ordre des opérations flottantes a été modifié par rapport à l'algorithme 8.6 : les résultats calculés par les algorithmes 8.6 et 8.18 sont rigoureusement identiques.

Algorithme 8.18. Implantation pratique de CompHornerK

fonction $\overline{r} = \mathsf{CompHornerK}(p_1, x, K)$
% Prologue de la boucle principale
$h_1 = p_{1,n}$
for $j = n-1 : -1 : n-K+2$
$e_1 = p_{1,j}$
for $i = 1 : 2^{n-j} - 1$
$[h_i, e_{2i}] = \mathsf{TwoProd}(h_i, x)$; $[h_i, e_{2i+1}] = \mathsf{TwoProd}(h_i, e_i)$
end
for $i = 1 : 2^{n-j+1} - 1$
$[h_i, h_{2i}] = \mathsf{TwoProd}(e_i, x)$; $[h_i, h_{2i+1}] = \mathsf{TwoProd}(h_i, e_i)$
end
end
% Boucle principale
for $j = n-3 : -1 : 0$
$e_1 = p_{1,j}$
for $i = 1 : 2^{K-1} - 1$
$[h_i, e_{2i}] = \mathsf{TwoProd}(h_i, x)$; $[h_i, e_{2i+1}] = \mathsf{TwoProd}(h_i, e_i)$
end

```
    for i = 2^{K-1} : 2^K - 1
      h_i = h_i ⊗ x ⊕ e_i
    end
  end
  % Sommation finale
  r̄ = SumK(h, 3)
```

8.7 Tests de performances

Pour ces tests, la double précision IEEE-754 est utilisée comme précision de travail : l'algorithme CompHornerK fournit donc une précision de calcul de l'ordre de $K \times 53$ bits.

Dans la première partie de ces tests, nous nous intéressons aux performances de l'algorithme CompHornerK (algorithme 8.18), en supposant que K est un paramètre d'entrée de l'algorithme. À titre de comparaison, nous utiliserons une implantation du schéma de Horner à l'aide de la bibliothèque multiprécision MPFR : nous désignerons par la suite par MPFRHornerK le schéma de Horner effectué à l'aide de la bibliothèque MPFR avec une précision de $K \times 53$ bits.

Nous présenterons ensuite les tests de performances que nous avons effectués avec l'algorithme CompHornerKBound (algorithme 8.12).

Finalement, nous nous intéressons à une version optimisée de l'algorithme CompHornerK dans le cas où $K = 4$: nous appellerons CompHorner4 l'algorithme obtenu en fixant $K = 4$ dans l'algorithme 8.18. Le fait de fixer le paramètre K permet d'effectuer certaines optimisations de code manuelles, et permet également au compilateur de produire un code mieux optimisé. Nous comparerons cette fois-ci les performances de CompHorner4 à celles d'une implantation du schéma de Horner à l'aide de la bibliothèque quad-double [35] : nous appellerons QDHorner le schéma de Horner basé sur cette bibliothèque, qui permet également de simuler une précision de travail quadruplée. Nos expériences illustrent l'efficacité de CompHorner4 face à QDHorner, en particulier sur l'architecture Itanium.

8.7.1 Performances pour un K quelconque

Nous comparons ici les performances de l'algorithme CompHornerK (algorithme 8.6) avec celles du schéma de Horner basé sur la bibliothèque multiprécision MPFR.

Rappelons que nous utilisons la double précision IEEE-754 comme précision de travail : cela signifie que l'algorithme CompHornerK fournit une précision de calcul de l'ordre de $K \times 53$ bits. Nous désignerons par la suite par MPFRHornerK le schéma de Horner effectué à l'aide de la bibliothèque MPFR avec une précision de $K \times 53$ bits. Nous effectuons ici nos mesures pour $K = 2, \ldots, 7$.

Précisons que les entrées de l'algorithme MPFRHornerK tel que nous l'avons implanté sont des flottants MPFR d'une précision de 53 bits : cela évite le coût des conversions de la double précision IEEE-754 vers le format des flottants MPFR lors des mesures de temps d'exécution. D'autre part, K est ici un paramètre d'entrée des algorithmes CompHornerK et MPFRHornerK : cela signifie que nous n'avons pas effectué les optimisations possibles lorsque K est fixée *a priori*. En particulier, aucun déroulage manuel de boucle n'est effectué dans le cas de CompHornerK.

Comme dans les mesures de performances des chapitres précédents, nous utilisons un

jeu de 39 polynômes générés aléatoirement, et dont le degré varie de 10 à 200 par pas de 5. Pour chaque degré considéré, on mesure le surcoût introduit par chacun des algorithmes CompHornerK et MPFRHornerK par rapport au schéma de Horner classique. Pour chaque environnement et chaque algorithme considérés, nous reportons sur les graphiques de la figure 8.5 la moyenne des surcoûts mesurés sur l'ensemble des 39 polynômes utilisés en fonction de K.

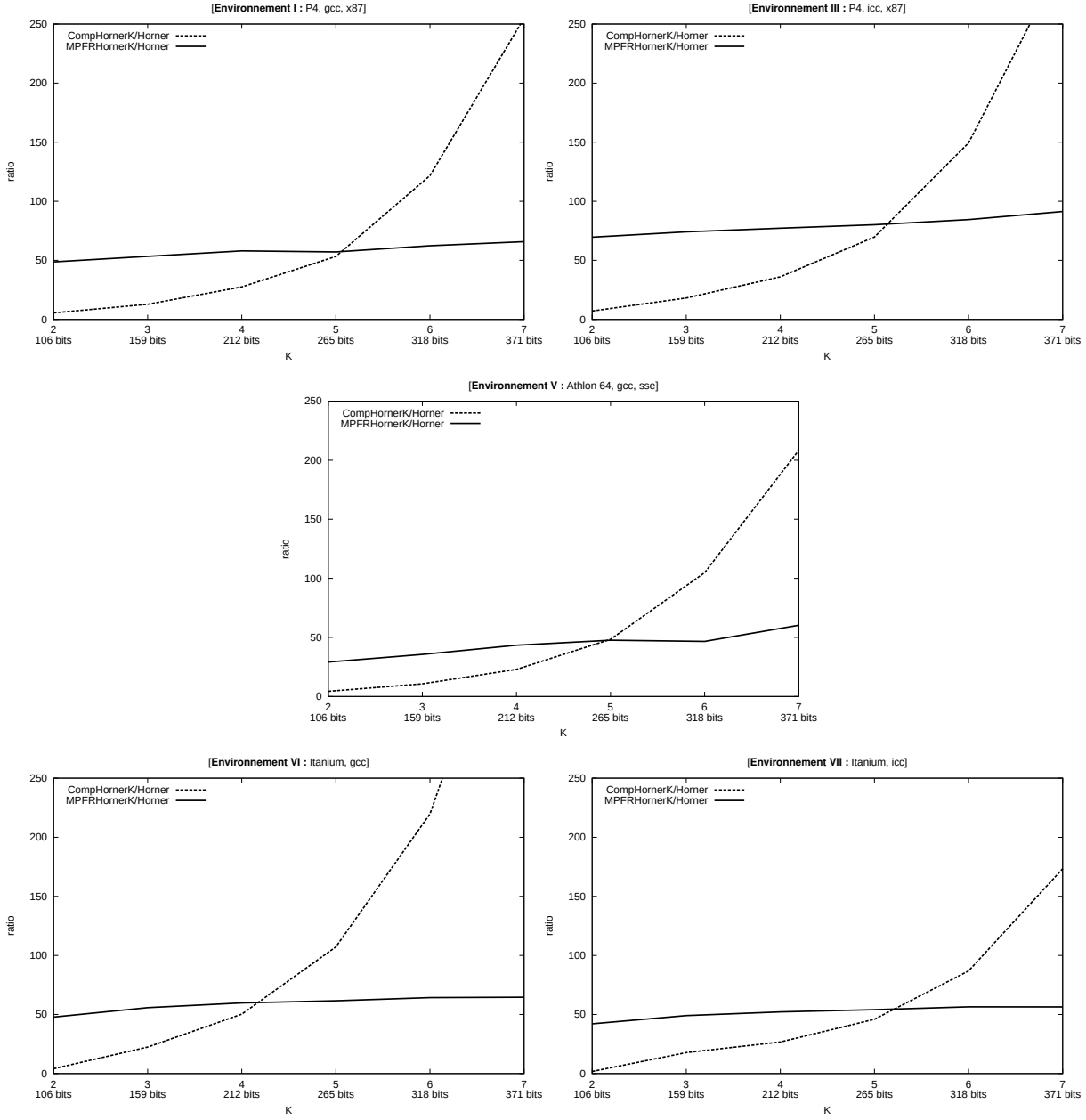


FIG. 8.5 – Surcoûts moyens mesurés pour CompHornerK, CompHornerKBound et MPFRHornerK par rapport à Horner.

Les environnement expérimentaux utilisés sont listés dans le tableau 5.1. Nous ne reportons pas ici les résultats obtenus dans les environnements (III) et (IV) car ils sont similaires à ceux obtenus dans les environnements (I) et (III).

Commençons par remarquer que la complexité en $\mathcal{O}(n2^K)$ de CompHornerK est clairement illustrée sur les graphiques de la figure 8.5. D'autre part, on constate que CompHornerK se montre dans nos expériences toujours plus rapide que MPFRHornerK tant que K est plus petit que 4. Ceci montre l'intérêt de CompHornerK pour doubler, tripler ou quadrupler la précision des calculs.

8.7.2 Surcoût introduit par la validation

Nous nous intéressons ici aux performances de CompHornerKBound (algorithme 8.12, que nous comparons à celle de l'algorithme non validé CompHornerK (algorithme 8.6). Nous utilisons les mêmes environnements expérimentaux et le même jeu de polynômes que dans la sous-section précédente.

Étant donné une valeur de K fixée entre 2 et 7, on mesure le ratio du temps d'exécution de CompHornerKBound sur celui de CompHornerK. Nous reportons sur la figure 8.6, pour chaque valeur de K considérée, la moyenne des surcoûts mesurés sur l'ensemble des 39 polynômes utilisés.

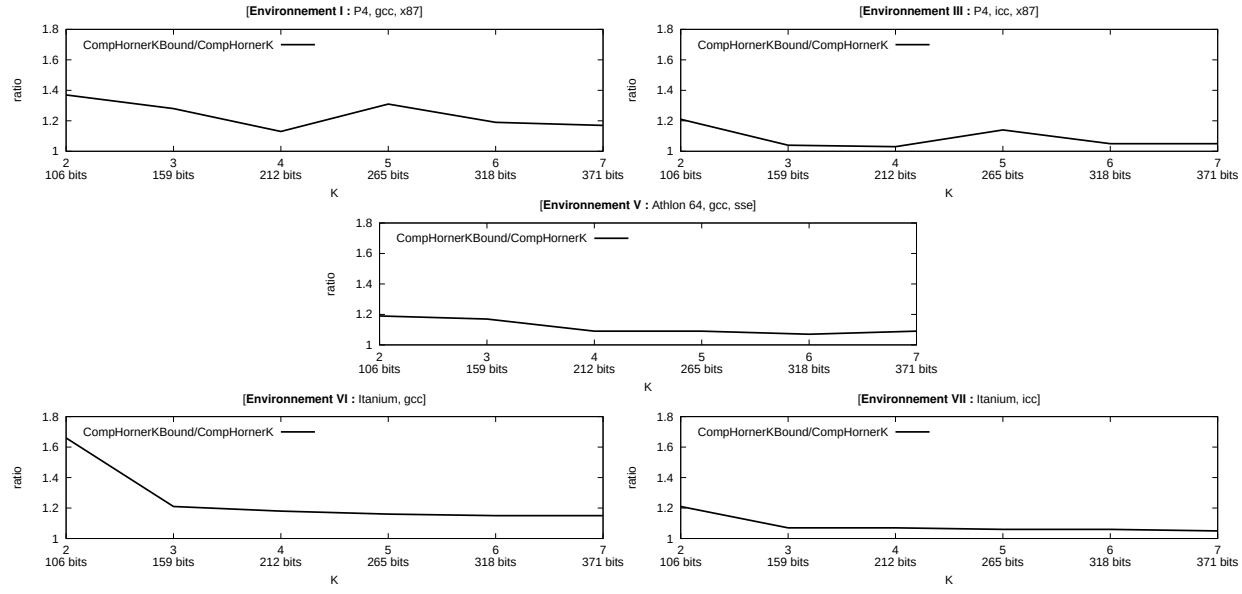


FIG. 8.6 – Surcoûts moyens mesurés introduits par CompHornerKBound par rapport à CompHornerK.

On constate sur la figure 8.6 que le calcul d'un résultat validé, à l'aide de CompHornerKBound, n'introduit qu'un faible surcoût par rapport à l'algorithme non validé CompHornerK. On retiendra en particulier que dans nos expériences, le surcoût moyen mesuré est toujours inférieur à 2.

8.7.3 Version optimisée pour $K = 4$

Ici, nous ciblons spécifiquement le cas $K = 4$, c'est à dire une précision de calcul quadruplée. Comme nous travaillons en double précision IEEE-754, nous simulerons donc à l'aide de CompHorner4 une précision de travail de l'ordre de 212 bits.

Nous ne formulerons pas dans les détails l'algorithme CompHorner4 que nous utilisons ici. Le code que nous utilisons dépend d'ailleurs de l'architecture sur laquelle les tests sont effectués. En particulier, sur l'architecture Itanium, les techniques décrites au chapitre 6 sont utilisées afin de tirer parti de l'instruction FMA disponible. Il suffit de savoir que le fait de fixer le paramètre K dans l'algorithme 8.18 permet d'effectuer certaines optimisations qui ne sont pas possibles lorsque K est variable (voir [33, chap. 4] et [16, 61, 3, 42]).

Nous comparons les performances de CompHorner4 à celles d'une implantation du schéma de Horner à l'aide de la bibliothèque quad-double[2] [35], que nous désignons sous le nom de QDHorner et qui permet également de simuler une précision de travail quadruplée. Notre implantation de QDHorner est basée sur les algorithmes proposés dans [35] pour l'arithmétique quad-double. Pour des raisons de performances, nous avons recodé ces algorithmes en C puis *inlinés* dans notre implantation de QDHorner qui a été compilée en utilisant les mêmes optimisations que pour CompHorner4.

Nous comparons également CompHorner4 à MPFRHorner4, qui est une implantation du schéma de Horner à l'aide de la bibliothèque MPFR, obtenue en fixant la précision de travail à 212 bits.

Comme dans les expériences précédentes, on utilise un jeu de 39 polynômes dont le degré varie de 10 à 200 par pas de 5. Pour chacun de ces polynômes, on mesure le ratio du temps d'exécution de CompHorner4 sur celui du schéma de Horner classique Horner. Pour chaque environnement expérimental, on reporte dans le tableau 8.1 la moyenne, sur l'ensemble des degrés envisagés, de ces surcoûts mesurés. On reporte également ces surcoûts moyens pour QDHorner et MPFRHorner4.

TAB. 8.1 – Surcoûts moyens mesurés pour CompHorner4, QDHorner et MPFRHorner4

environnement		$\frac{\text{CompHorner4}}{\text{Horner}}$	$\frac{\text{QDHorner}}{\text{Horner}}$	$\frac{\text{MPFRHorner4}}{\text{Horner}}$
(I)	P4, gcc, x87	20.2	32.6	48.7
(II)	P4, gcc, sse	22.5	36.4	55.7
(III)	P4, icc, x87	21.0	44.1	51.6
(IV)	P4, icc, sse	23.1	39.4	58.7
(V)	Ath64, gcc, sse	21.4	34.9	42.0
(VI)	Itanium, gcc	18.7	39.4	58.5
(VII)	Itanium, icc	4.8	41.0	51.8

On constate que CompHorner4 s'exécute toujours significativement plus rapidement que QDHorner et MPFRHorner. On notera en particulier que CompHorner4 s'exécute environ 8 fois plus rapidement que QDHorner dans l'environnement (VII), c'est à dire sur l'architecture Itanium et avec le compilateur Intel.

8.8 Conclusions

Dans ce chapitre, nous avons présenté l'algorithme compensé CompHornerK (algorithme 8.6) : nous avons démontré que le résultat compensé calculé par cet algorithme est aussi précis

[2]Bibliothèque disponible à l'adresse `http://crd.lbl.gov/~dhbailey/mpdist/`

que s'il avait été calculé par le schéma de Horner en K fois la précision de travail ($\mathbf{u}^K$), puis arrondi vers le précision de travail $\mathbf{u}$.

Nous avons également présenté une version validée de CompHornerK, l'algorithme CompHornerKBound (algorithme 8.12). Rappelons en effet que cet algorithme permet de calculer à la fois le résultat compensé $\overline{r} = \mathsf{CompHornerK}(p, x)$, ainsi qu'une borne d'erreur *a posteriori* pour ce résultat compensé.

Tout comme le schéma de Horner compensé présenté dans les chapitres précédents, les algorithmes CompHornerK et CompHornerKBound ne nécessitent que des opérations flottantes élémentaires, effectuées à une précision de travail $\mathbf{u}$ fixée, dans le mode d'arrondi au plus proche : il s'agit donc d'algorithmes facilement portables, dans tout environnement où l'arithmétique flottante est conforme à la norme IEEE-754.

Nos tests de performances montrent que l'algorithme CompHornerK est une alternative efficace aux solutions génériques pour doubler, tripler ou quadrupler la précision de l'évaluation polynomiale. Par solutions génériques, nous désignons ici les bibliothèques de calcul flottant multiprécision, comme MPFR, ou encore la bibliothèque quad-double. Ces performances justifient l'intérêt pratique de l'algorithme CompHornerK lorsqu'il est nécessaire d'augmenter la précision des calculs d'un facteur 2, 3 ou 4.

Chapitre 9

Résolution compensée de systèmes triangulaires

Plan du chapitre : En Section 9.1, nous exposons le principe du calcul d'une solution compensée d'un système triangulaire, en montrant le lien qu'il existe avec la méthode bien connue du raffinement itératif. On constate en particulier que la seule latitude dont on dispose quant au calcul d'une solution compensée réside dans le choix d'une méthode de calcul du résidu. En Section 9.2, nous rappelons les résultats connus sur la méthode de substitution pour la résolution de systèmes triangulaires, et montrons que le problème du calcul du résidu associé à une solution calculée par substitution est toujours mal conditionné. La Section 9.3 est dédiée à une analyse d'erreur pour le calcul d'une solution compensée à un système triangulaire : nous obtenons un résultat générique, en supposant uniquement que le résidu est calculé en précision doublée, et permettant de borner l'erreur entachant la solution compensée. Ce résultat est ensuite utilisé dans les Section 9.4 et 9.5, dans lesquels nous envisageons deux méthodes de calcul du résidu. Dans la section Section 9.5, nous montrons en effet que le résidu peut être calculé en fonction des erreurs d'arrondi élémentaires générées par l'algorithme de substitution. Nous en déduisons un premier algorithme compensé pour la résolution de systèmes triangulaires, CompTRSV (algorithme 9.9). En Section 9.5, nous envisageons le cas où le résidu est calculé à l'aide de l'algorithme Dot2 de Ogita, Rump et Oishi [70], permettant le calcul de produits scalaires en précision doublée : on obtient ainsi un second algorithme compensé, CompTRSV2 (algorithme 9.14). Nous montrons que CompTRSV et CompTRSV2 peuvent être formulés de façon quasi-équivalente, à l'ordre de quelques opérations flottantes prés. Dans nos expériences numériques de la section Section 9.6, nous nous concentrerons donc sur l'algorithme CompTRSV. La Section 9.7 est dédiée aux tests de performances.

9.1 Introduction

Soit $Tx = b$ un système triangulaire d'équations linéaires, avec $T \in \mathbf{F}^{n\times n}$ et $b \in \mathbf{F}^{n\times 1}$. Pour fixer les idées, on supposera par la suite que T est une matrice triangulaire inférieure. On supposera en outre que la matrice T est non singulière. On considère donc une système

linéaire de la forme

$$\begin{pmatrix} t_{1,1} & & \\ \vdots & \ddots & \\ t_{n,1} & \cdots & t_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}.$$

La solution exacte $x = (x_1, \ldots, x_n)^T$ de ce système d'équations peut être obtenue à l'aide de la formule de substitution,

$$x_k = \frac{1}{t_{k,k}} \left(b_k - \sum_{i=1}^{k-1} t_{k,i} x_i \right), \quad \text{pour} \quad k = 1 : n, \tag{9.1}$$

qui permet de calculer successivement $x_1, x_2, \ldots, x_n$.

Supposons maintenant que les coefficients du système triangulaire $Tx = b$ sont des flottants. Comme à l'accoutumée on travaille en arithmétique flottante à la précision $\mathbf{u}$, en arrondi au plus proche. On supposera toujours implicitement dans ce chapitre qu'aucun dépassement de capacité n'intervient au cours des calculs.

En utilisant la relation (9.1) on écrit ci-dessous l'algorithme classique de substitution pour la résolution des systèmes triangulaires, en fixant bien entendu un ordre particulier pour l'évaluation des opérations arithmétiques. Cet algorithme nécessite n^2 opérations flottantes.

Algorithme 9.1.

function $\widehat{x} = \mathsf{TRSV}(T, b)$
  for $k = 1 : n$
    $\widehat{s}_{k,0} = b_k$
    for $i = 1 : k-1$
      $\widehat{p}_{k,i} = t_{k,i} \otimes \widehat{x}_i$
      $\widehat{s}_{k,i} = \widehat{s}_{k,i-1} \ominus \widehat{p}_{k,i}$
    end
    $\widehat{x}_k = \widehat{s}_{k,k-1} \oslash t_{k,k}$
  end

L'algorithme de substitution est connu pour sa stabilité [39, chap. 8][36]. De plus, l'erreur directe entachant la solution calculée $\widehat{x} = \mathsf{TRSV}(T, b)$ est souvent nettement plus petite que ce qu'indique l'analyse d'erreur [39, p. 140]. Néanmoins, la précision de la solution $\widehat{x}$ calculée par substitution vérifie

$$\frac{\|\widehat{x} - x\|_\infty}{\|\widehat{x}\|_\infty} \leq n\mathbf{u}\,\mathrm{cond}(T, x) + \mathcal{O}(\mathbf{u}^2), \tag{9.2}$$

où $\mathrm{cond}(T, x)$ désigne le conditionnement de Skeel du système triangulaire qui sera explicité plus loin (voir Section 9.2). Indiquons simplement que $\mathrm{cond}(T, x)$ dépend des coefficients du système triangulaire considéré, et peut prendre des valeurs arbitrairement grandes. La solution $\widehat{x}$ calculée en arithmétique flottante peut donc être arbitrairement moins précise que l'unité d'arrondi $\mathbf{u}$, en particulier lorsque $\mathrm{cond}(T, x)$ est grand devant $\mathbf{u}^{-1}$, ce qui sera effectivement observé dans nos expériences numériques.

Comment compenser l'effet des erreurs d'arrondi sur la solution approchée $\widehat{x}$? Définissons le vecteur $c \in \mathbf{R}^{n\times 1}$ comme étant l'erreur directe affectant la solution calculée $\widehat{x}$, c'est à dire

$$c = x - \widehat{x}.$$

Comme dans le cas du schéma de Horner compensé, il faut calculer une approximation $\widehat{c}$ du vecteur c, puis un résultat compensé $\overline{x} = \widehat{x} \oplus \widehat{c}$. Par définition du vecteur c, on a $x = \widehat{x} + c$. On s'attend donc à ce que le résultat compensé $\overline{x}$ soit plus précis que le résultat initial $\widehat{x}$. Reste à déterminer comment calculer le terme correctif approché $\widehat{c}$. Puisque $c = x - \widehat{x}$ et $x = T^{-1}b$, on a

$$c = T^{-1}(b - T\widehat{x}).$$

Définissons le vecteur $r \in \mathbf{R}^{n\times 1}$ par

$$r = b - T\widehat{x}.$$

En utilisant la terminologie du raffinement itératif, on dira que r est le résidu associé à la solution approchée $\widehat{x}$ du système $Tx = b$.

Le raffinement itératif est une méthode bien connue pour améliorer la qualité d'une solution approchée $\widehat{x}$ d'un système linéaire $Ax = b$, qui peut être résumée comme suit [39, p. 232].

1. Calculer une solution approchée $\widehat{x}$ du système $Ax = b$.
2. Calculer une valeur approchée $\widehat{r}$ du résidu $r = b - A\widehat{x}$.
3. Calculer une solution approchée $\widehat{c}$ du système $Ac' = \widehat{r}$.
4. Mettre à jour $\widehat{x}$, en calculant $\widehat{x} = \widehat{x} \oplus \widehat{c}$.

   Reprendre si nécessaire à l'étape 2.

Il est à noter que si le résidu approché $\widehat{r}$ et le terme correctif approché $\widehat{c}$ étaient calculés exactement, alors $\widehat{x} + \widehat{c}$ serait la solution exacte du système linéaire considéré. Ce ne sera généralement pas le cas en pratique, mais si $\widehat{r}$ et $\widehat{c}$ sont calculés suffisamment précisément, on peut s'attendre à ce que la qualité de la solution soit effectivement améliorée. Le comportement numérique du processus de raffinement itératif a été largement étudié dans la littérature [89, 63, 8, 38, 25]. Le principal résultat connu concerne le cas où le système $Ax = b$ est résolu par élimination Gaussienne à la précision courante ($\mathbf{u}$), et le résidu calculé dans une précision de travail doublée ($\mathbf{u}^2$). Higham résume ce résultat de la façon suivante [39, p. 232].

> Traditionally, iterative refinement is used with Gaussian elimination (GE), and r is computed in extended precision before being rounded to working precision. [...] The behaviour of iterative refinement for GE is usually summarized as follows : if double the working precision is used in the computation of r, and A is not too ill conditioned, then the iteration produces a solution correct to working precision and the rate of convergence depends on the condition number of A.

Notons que le raffinement itératif avec calcul du résidu en précision doublée peut aujourd'hui être implanté de façon portable sur les plates-formes disposant d'une arithmétique IEEE-754 à l'aide des *Extended and Mixed precision BLAS* [56, 57].

Replaçons nous dans le cadre de la résolution compensée du système triangulaire $Tx = b$ qui nous intéresse ici. Le fait de calculer un terme correctif $\widehat{c}$, puis une solution compensée $\overline{x}$ s'avère équivalent à l'utilisation d'une étape de raffinement itératif pour améliorer la précision du résultat initial $\widehat{x}$. En effet, pour obtenir le résultat compensé $\overline{x}$, on calcule successivement :

1. la solution initiale $\widehat{x} = \mathsf{TRSV}(T, b)$,
2. une valeur approchée $\widehat{r}$ du résidu $r = b - T\widehat{x}$,
3. un terme correctif approché $\widehat{c} = \mathsf{TRSV}(T, \widehat{r})$,
4. le résultat compensé $\overline{x} = \widehat{x} \oplus \widehat{c}$.

Le seul degré de liberté dans la procédure de calcul de la solution compensé $\overline{x}$ énoncée ci-dessus concerne le calcul du résidu approché $\widehat{r}$. Nous envisageons ici deux possibilités.

Nous proposons tout d'abord une méthode de calcul du résidu basée sur l'utilisation des transformations exactes pour les opérations arithmétiques élémentaires. Nous montrons en effet que le résidu $r = b - Tx$ peut être exprimé exactement en fonction des erreurs d'arrondis générées par les opérations arithmétiques intervenant dans l'algorithme de substitution. De ce résultat, on déduit facilement une méthode de calcul d'un résidu approché $\widehat{r}$: nous montrerons que celui-ci est aussi précis que le résidu calculé en précision doublée. On appellera CompTRSV l'algorithme compensé de résolution de système qui résulte de cette méthode de calcul du résidu.

À titre de comparaison, nous étudierons ensuite le cas où le résidu est calculé à l'aide de l'algorithme Dot2, qui est un algorithme de l'état de l'art connu pour son efficacité, et permettant le calcul de produits scalaires en précision doublée [70]. Dot2 permet donc également de calculer en précision doublée le résidu associé à la solution approchée d'un système triangulaire. Nous montrerons que le calcul du résidu à l'aide de Dot2 est en fait quasiment équivalent à la méthode de calcul du résidu à l'aide des transformations exactes proposé précédemment : en effet, ces deux méthodes de calcul d'un résidu approché différent seulement par l'ordre de quelques opérations arithmétiques, et présentent des bornes d'erreurs similaires. L'algorithme résultant du calcul du résidu à l'aide de Dot2 sera appelé CompTRSV2.

Rappelons que le seul degré de liberté dont nous disposons, quant au calcul du résultat compensé $\overline{r}$, réside dans le choix d'une méthode de calcul du résidu approché $\widehat{r}$. Les méthodes de calculs du résidu sont quasiment identiques dans CompTRSV et CompTRSV2. De plus, nous obtenons des bornes d'erreurs similaires pour la précision de la solution calculée par ces deux algorithmes, et il est difficile de mettre en évidence en pratique une différence de précision entre les deux méthodes. Nous considérons donc à nouveau ces deux algorithmes compensés comme équivalents, et nous nous concentrerons uniquement par la suite sur l'étude de CompTRSV.

Nos expériences numériques semblent montrer que la solution compensée $\overline{x}$ calculée par CompTRSV est aussi précise que si elle avait été calculée en précision doublée, avec un arrondi final vers la précision courante. Nous verrons en particulier que la solution compensée présente dans nos expériences numériques la même précision effective que la solution obtenue à l'aide de la fonction BLAS_dtrsv_x de la bibliothèque XBLAS [57] : La fonction BLAS_dtrsv_x calcule une solution approchée à un système triangulaire par l'algorithme de substitution en arithmétique double-double, simulant ainsi un doublement

de la précision des calculs, puis arrondi cette solution vers la double précision IEEE-754. Comme nous le verrons plus en détail par la suite, une solution approchée $\widehat{x}_d$ au système triangulaire $Tx = b$, calculée en précision doublée, puis arrondie vers la précision courante $\mathbf{u}$ satisfait à une borne d'erreur relative de la forme suivante,

$$\frac{\|\widehat{x}_d - x\|_\infty}{\|x\|_\infty} \leq \mathbf{u} + \mathcal{O}(\mathbf{u}^2)\,\mathrm{cond}(T, x) + \mathcal{O}(\mathbf{u}^3). \tag{9.3}$$

Comme dans le cas du schéma de Horner compensé (voir chapitre 4), le facteur en $\mathcal{O}(\mathbf{u}^2)$ devant le conditionnement de Skeel du système triangulaire reflète la précision doublée des calculs intermédiaires ; le terme $\mathbf{u}$ reflète quant à lui l'arrondi final vers la précision courante.

Nous ne sommes pas parvenu à démontrer que la solution calculée par CompTRSV est aussi précise que si elle avait été calculée avec une précision doublée pour les calculs intermédiaires : en d'autre termes, nous ne sommes pas parvenu à obtenir une borne sur l'erreur relative entachant la solution compensée $\widehat{x}$ de la même forme que l'inégalité (9.3). Néanmoins, au travers du lemme 9.4, nous proposons une méthode générique pour obtenir une borne *a priori* sur l'erreur entachant la solution compensée. Ce résultat appliqué à l'algorithme CompTRSV, donne une borne d'erreur relative de la forme,

$$\frac{\|\widehat{x}_d - x\|_\infty}{\|x\|_\infty} \leq \mathbf{u} + \mathcal{O}(\mathbf{u}^2)\,\mathrm{K}(T, x) + \mathcal{O}(\mathbf{u}^3), \tag{9.4}$$

où $\mathrm{K}(T, x)$ est un facteur qui sera défini en Section 9.3, et vérifiant $\mathrm{cond}(T, x) \leq \mathrm{K}(T, x)$. Au travers de nos expériences numériques, nous verrons que l'on peut s'attendre à ce que $\mathrm{K}(T, x)$ diffère en pratique relativement peu de $\mathrm{cond}(T, x)$, ce qui nous permettra d'interpréter expérimentalement la borne d'erreur (9.4).

9.2 Algorithme de substitution et résidu associé

Dans cette section, nous rappelons les résultats bien connus qui nous serons utiles à propos de l'algorithme de substitution en arithmétique flottante. Nous montrons également que le calcul du résidu, pour une solution approchée calculée par substitution d'un système triangulaire, est nécessairement un problème mal conditionné à la précision $\mathbf{u}$.

9.2.1 Précision de la solution calculée

Higham propose le théorème suivant, valable lorsque la formule de substitution est utilisée en arithmétique flottante, quel que soit l'ordre d'évaluation des opérations arithmétiques [39, p. 142].

Théorème 9.2. *On considère le système triangulaire $Tx = b$ avec $T \in \mathbf{F}^{n\times n}$ non singulière et $b \in \mathbf{F}^{n\times 1}$. Soit $\widehat{x}$ une solution approchée de ce système calculé en arithmétique flottante par substitution, dans un ordre quelconque. Alors la solution calculée $\widehat{x}$ satisfait*

$$(T + \Delta T)\widehat{x} = b, \quad \textit{avec} \quad |\Delta T| \leq \gamma_n |T|.$$

Le théorème précédent est un résultat de nature inverse : la solution calculée $\widehat{x}$ est exprimée comme solution exacte d'un système perturbé $(T+\Delta T)\widehat{x} = b$. Higham déduit ensuite du théorème 9.2 la borne suivante sur l'erreur relative entachant la solution calculée [39, p. 142],

$$\frac{\|\widehat{x}-x\|_\infty}{\|\widehat{x}\|_\infty} \leq \frac{\gamma_n \operatorname{cond}(T,x)}{1-\gamma_n \operatorname{cond}(T)}, \tag{9.5}$$

où $\operatorname{cond}(T,x)$ et $\operatorname{cond}(T)$ sont deux nombres de conditionnement introduits par Skeel. Ceux-ci sont définis respectivement par

$$\operatorname{cond}(T,x) = \frac{\||T^{-1}||T||x|\|_\infty}{\|x\|_\infty}, \tag{9.6}$$

et

$$\operatorname{cond}(T) = \operatorname{cond}(T,e) = \||T^{-1}||T|\|_\infty, \tag{9.7}$$

avec $e = (1,\ldots,1)^T \in R^{n\times 1}$. Il faut bien entendu supposer $\gamma_n \operatorname{cond}(T) < 1$ pour que l'inégalité (9.5) ait un sens. En négligeant les termes en $\mathcal{O}(\mathbf{u}^2)$, on obtient ainsi

$$\frac{\|\widehat{x}-x\|_\infty}{\|\widehat{x}\|_\infty} \leq n\mathbf{u} \operatorname{cond}(T,x) + \mathcal{O}(\mathbf{u}^2). \tag{9.8}$$

Cette dernière inégalité montre que la borne (9.5) sur l'erreur relative entachant la solution calculée dépendant essentiellement du conditionnement $\operatorname{cond}(T,x)$.

Supposons maintenant que l'on calcule une solution approchée $\widehat{x}_d$ au système $Tx = b$ par substitution en précision doublée $\mathbf{u}^2$, avec un arrondi final vers la précision de travail $\mathbf{u}$. On note ici $\overline{\gamma}_k = k\mathbf{u}^2/(1-k\mathbf{u}^2)$. En supposant $\overline{\gamma}_n \operatorname{cond}(T) < 1$, la solution $\widehat{x}$ du système $Tx = b$ calculée par substitution en précision doublée satisfait

$$\frac{\|\widehat{x}-x\|_\infty}{\|x\|_\infty} \leq \frac{\overline{\gamma}_n \operatorname{cond}(T,x)}{1-\overline{\gamma}_n \operatorname{cond}(T)}.$$

De plus, comme $\widehat{x}_d$ est l'arrondi de $\widehat{x}$ à la précision courante,

$$\frac{\|\widehat{x}_d-x\|_\infty}{\|x\|_\infty} \leq \mathbf{u} + (1+\mathbf{u})\frac{\overline{\gamma}_n \operatorname{cond}(T,x)}{1-\overline{\gamma}_n \operatorname{cond}(T)},$$

ce qui donne, en négligeant les termes en $\mathcal{O}(\mathbf{u}^3)$,

$$\frac{\|\widehat{x}_d-x\|_\infty}{\|x\|_\infty} \leq \mathbf{u} + n\mathbf{u}^2 \operatorname{cond}(T,x) + \mathcal{O}(\mathbf{u}^3). \tag{9.9}$$

Les bornes d'erreurs obtenues pour la solution approchée au système $Tx = b$ calculée par substitution compensée seront par la suite comparées à cette borne d'erreur (9.9).

9.2.2 Nombre de conditionnement pour le calcul du résidu

Le but de cette sous-section est de montrer que le calcul du résidu $r = b - T\widehat{x}$ est un problème mal conditionné, indépendamment de $\operatorname{cond}(T)$ et de $\operatorname{cond}(T,x)$. Insistons sur le fait que le résultat que nous allons démontrer ne s'applique qu'au cas des systèmes triangulaires, et en supposant que la solution approchée $\widehat{x}$ du système $Tx = b$ est obtenue

par substitution. Précisions également que nous ne connaissons pas de résultat comparable dans le cas de systèmes linéaires quelconques. Nous introduisons ici un nombre de conditionnement pour le calcul de r, défini par

$$\mathrm{cond}(r) = \frac{\||b| + |T||\widehat{x}|\|_\infty}{\|r\|_\infty}. \tag{9.10}$$

Afin d'illustrer le rôle joué par $\mathrm{cond}(r)$, considérons le cas où $\widehat{r}$ est calculé en précision de travail $\mathbf{u}$. L'erreur directe entachant le résidu calculé $\widehat{r}$ est alors majorée comme suit (voir [39, p. 233]),

$$|\widehat{r} - r| \leq \gamma_{n+1}(|b| + |T||\widehat{x}|).$$

En supposant $\|r\|_\infty \neq 0$, on a donc

$$\frac{\|\widehat{r} - r\|_\infty}{\|r\|_\infty} \leq \gamma_{n+1}\,\mathrm{cond}(r). \tag{9.11}$$

Cette borne sur l'erreur relative entachant le résidu calculé $\widehat{r}$ nous permet de justifier le fait que nous ayons défini $\mathrm{cond}(r)$ comme un nombre de conditionnement pour le calcul de r. La proposition suivante fournit un minorant pour $\mathrm{cond}(r)$, qui ne dépend ni de $\mathrm{cond}(T)$, ni de $\mathrm{cond}(T, x)$.

Proposition 9.3. *En supposant que $\widehat{x}$ soit une solution approchée du système triangulaire $Tx = b$ calculée par substitution, indépendamment de l'ordre l'ordre d'évaluation des opérations arithmétiques, on a*

$$\mathrm{cond}(r) \geq \gamma_n^{-1}. \tag{9.12}$$

La relation (9.12) montre que le conditionnement pour le calcul du résidu est toujours supérieur à $\gamma_n^{-1} \approx \mathbf{u}^{-1}$: le calcul du résidu est donc un problème mal conditionné lorsque l'on travaille à la précision $\mathbf{u}$.

En particulier, lorsque le résidu est calculé en précision courante $\mathbf{u}$, la borne (9.11) ne permet pas de garantir une erreur relative inférieure à 1 pour le résidu approché $\widehat{r}$. Ainsi, la proposition 9.3 justifie, dans le cas particulier de la résolution de systèmes triangulaires par substitution, la nécessité de calculer le résidu dans une précision au moins double de la précision de travail $\mathbf{u}$.

Preuve. Comme $\widehat{x}$ est une solution approchée du système triangulaire $Tx = b$ obtenue par substitution, on a $b = (T + \Delta T)\widehat{x}$, avec $|\Delta T| \leq \gamma_n |T|$. Donc

$$|r| = |b - T\widehat{x}| = |(T + \Delta T)\widehat{x} - T\widehat{x}| = |\Delta T\widehat{x}| \leq \gamma_n |T||\widehat{x}|,$$

et, $\|r\|_\infty \leq \gamma_n \||T||\widehat{x}|\|_\infty$. D'où

$$\frac{\||b| + |T||\widehat{x}|\|_\infty}{\|r\|_\infty} \geq \gamma_n^{-1} \frac{\||b| + |T||\widehat{x}|\|_\infty}{\||T||\widehat{x}|\|_\infty} \geq \gamma_n^{-1},$$

ce qui démontre la proposition. ■

9.3 Compensation de l'algorithme de substitution

Cette section est dédiée à la preuve du lemme 9.4 ci-dessous. Dans ce lemme, nous rappelons la méthode utilisée pour compenser les erreurs d'arrondi générées lors du calcul d'une solution approchée du système $Tx = b$ par substitution. Mais surtout, ce lemme permet d'obtenir une borne *a priori* sur l'erreur entachant la solution compensée $\overline{x}$, et ce quasi-indépendamment de la méthode utilisée pour le calcul du résidu approché $\widehat{r}$. La seule hypothèse que nous formulons ici est que l'erreur $|\widehat{r} - r|$ entachant le résidu approché $\widehat{r}$ satisfait

$$|\widehat{r} - r| \leq \mathbf{u}|r| + \mathcal{O}(\mathbf{u}^2),$$

ce qui est vrai en particulier lorsque $\widehat{r}$ est calculé en précision doublée. Ce lemme sera utilisé dans les sections suivantes pour démontrer les théorèmes 9.10 et 9.15.

Lemme 9.4. *Soit $Tx = b$ un système triangulaire d'équations linéaires, avec $T \in \mathbf{F}^{n\times n}$ une matrice triangulaire inférieure telle que $\gamma_n \operatorname{cond}(T) < 1$, et $b \in \mathbf{F}^{n\times 1}$. Soient*

- *$\widehat{x}$ une solution approchée du système triangulaire $Tx = b$ calculée par substitution,*
- *$\widehat{r}$ une valeur approchée du résidu $r = b - T\widehat{x}$, calculée par une méthode quelconque,*
- *$\widehat{c}$ une solution approchée du système $Tc' = b$ calculée par substitution,*
- *$\overline{x} = \widehat{r} \oplus \widehat{c}$ la solution compensée du système $Tx = b$.*

Si de plus on suppose $|\widehat{r} - r| \leq \mathbf{u}|r| + \mathcal{O}(\mathbf{u}^2)$, alors l'erreur directe entachant la solution compensée $\overline{x}$ est majorée comme suit,

$$|\overline{x} - x| \leq \mathbf{u}|x| + |T^{-1}||r - \widehat{r}| + \gamma_n^2(|T^{-1}||T|)^2|x| + \mathcal{O}(\mathbf{u}^3). \tag{9.13}$$

Rappelons que le seul degré de liberté dont nous disposons, quant au calcul du résultat compensé $\overline{r}$, réside dans le choix d'une méthode de calcul du résidu approché $\widehat{r}$. Étant donné un algorithme de calcul de $\widehat{r}$ permettant d'assurer que l'hypothèse $|\widehat{r} - r| \leq \mathbf{u}|r| + \mathcal{O}(\mathbf{u}^2)$ est vérifiée, le lemme 9.4 fournit immédiatement une borne sur l'erreur directe entachant la solution compensée $\overline{r}$: il suffit en effet d'obtenir une majoration de $|\widehat{r} - r|$ dans laquelle seuls les termes en $\mathcal{O}(\mathbf{u}^3)$ sont négligés, et de reporter cette majoration dans l'inégalité (9.13).

Preuve du lemme 9.4. Comme $\overline{x} = \widehat{x} \oplus \widehat{c}$, on a

$$\begin{aligned} |\overline{x} - x| &\leq |\overline{x} - (\widehat{x} + \widehat{c})| + |(\widehat{x} + \widehat{c}) - x| \\ &\leq \mathbf{u}|\widehat{x} + \widehat{c}| + |(\widehat{x} + \widehat{c}) - x|. \end{aligned}$$

Puisque $x = \widehat{x} + c$, il vient $|\overline{x} - x| \leq \mathbf{u}|x + \widehat{c} - c| + |\widehat{c} - c|$, d'où

$$|\overline{x} - x| \leq \mathbf{u}|x| + (1 + u)|\widehat{c} - c|. \tag{9.14}$$

Comme la solution approchée $\widehat{x}$ du système triangulaire $Tx = b$ est obtenue par substitution, elle satisfait

$$(T + \Delta T)\widehat{x} = b, \quad \text{avec} \quad |\Delta T| \leq \gamma_n|T|.$$

De plus, l'hypothèse $\gamma_n \operatorname{cond}(T) = \gamma_n \||T^{-1}||T|\|_\infty < 1$ nous assure que la matrice $(T + \Delta T)$ est non singulière, et l'on peut écrire

$$(T + \Delta T)^{-1} = \big(T(I + T^{-1}\Delta T)\big)^{-1} = (I + T^{-1}\Delta T)^{-1}T^{-1} = (I + F)T^{-1},$$

avec $|F| \leq \gamma_n |T^{-1}||T| + \mathcal{O}(\mathbf{u}^2)$ (voir [39, p. 233][27, p. 58]). On en déduit que

$$|\widehat{x} - x| = |(I+F)T^{-1}b - x| = |Fx| \leq \gamma_n |T^{-1}||T||x| + \mathcal{O}(\mathbf{u}^2).$$

D'autre part, $\widehat{c}$ est solution approchée du système triangulaire $Tc' = b$, également calculée par substitution, donc :

$$\begin{gathered}(T+\Delta T')\widehat{c} = \widehat{r}, \quad \text{avec} \quad |\Delta T'| \leq \gamma_n |T|, \quad \text{et} \\ (T+\Delta T')^{-1} = (I+F')T^{-1}, \quad \text{avec} \quad |F'| \leq \gamma_n |T^{-1}||T| + \mathcal{O}(\mathbf{u}^2).\end{gathered}$$

Passons maintenant à la majoration de l'erreur directe entachant le terme correctif calculé $\widehat{c}$. Comme $|c - \widehat{c}| = |c - T^{-1}\widehat{r} + T^{-1}\widehat{r} - \widehat{c}|$, on majore l'erreur directe $|c - \widehat{c}|$ comme suit,

$$|c - \widehat{c}| \leq |c - T^{-1}\widehat{r}| + |T^{-1}\widehat{r} - \widehat{c}|. \tag{9.15}$$

Le premier terme dans l'inégalité précédente prend en compte l'erreur commise en approchant le résidu exact r par le résidu approché $\widehat{r}$. Le second terme prend lui en compte l'erreur commise en approchant $\widehat{c}$ par $\mathsf{TRSV}(T, \widehat{r})$ plutôt que par $T^{-1}\widehat{r}$. D'une part, comme $c = T^{-1}r$, on a

$$|c - T^{-1}\widehat{r}| = |T^{-1}(r - \widehat{r})| \leq |T^{-1}||r - \widehat{r}|.$$

D'autre part, comme $\widehat{c} = (I+F')T^{-1}\widehat{r}$, on a $|T^{-1}\widehat{r} - \widehat{c}| = |F'T^{-1}\widehat{r}|$. En écrivant $\widehat{r} = r + \widehat{r} - r$, il vient

$$|T^{-1}\widehat{r} - \widehat{c}| \leq |F'T^{-1}r| + |F'T^{-1}(\widehat{r} - r)|.$$

Or, $T^{-1}r = x - \widehat{x}$, donc

$$|T^{-1}\widehat{r} - \widehat{c}| \leq |F'||\widehat{x} - x| + |F'||T^{-1}||\widehat{r} - r|.$$

En reportant les majorations obtenues pour $|c - T^{-1}\widehat{r}|$ et $|T^{-1}\widehat{r} - \widehat{c}|$ dans l'inégalité (9.15), on obtient

$$|c - \widehat{c}| \leq |T^{-1}||r - \widehat{r}| + |F'||\widehat{x} - x| + |F'||T^{-1}||\widehat{r} - r|.$$

Comme $|\widehat{x} - x| \leq \gamma_n |T^{-1}||T||x| + \mathcal{O}(\mathbf{u}^2)$ et $|F'| \leq \gamma_n |T^{-1}||T| + \mathcal{O}(\mathbf{u}^2)$, on a

$$|c - \widehat{c}| \leq |T^{-1}||r - \widehat{r}| + \gamma_n^2 (|T^{-1}||T|)^2 |x| + |F'||T^{-1}||\widehat{r} - r| + \mathcal{O}(\mathbf{u}^3). \tag{9.16}$$

Montrons de plus que le terme $|F'||T^{-1}||\widehat{r} - r|$ peut être négligé d'après l'hypothèse $|\widehat{r} - r| \leq \mathbf{u}|r| + \mathcal{O}(\mathbf{u}^2)$. Notons que

$$|r| = |T||\widehat{x} - x| \leq \gamma_n |T||T^{-1}||T||x| + \mathcal{O}(\mathbf{u}^2).$$

On en déduit

$$|\widehat{r} - r| \leq \mathbf{u}\gamma_n |T||T^{-1}||T||x| + \mathcal{O}(\mathbf{u}^3).$$

Puisque $|F'| \leq \gamma_n |T^{-1}||T| + \mathcal{O}(\mathbf{u}^2)$, le terme $|F'||T^{-1}||\widehat{r} - r|$ dans la majoration (9.16) peut donc être négligé. On obtient ainsi le résultat annoncé. ■

Indiquons que le lemme 9.4 constitue le résultat le plus satisfaisant dont nous disposons, au sens où nous ne sommes pas parvenu à obtenir de majoration *a priori* plus fine de l'erreur entachant la solution compensée $\overline{x}$.

Il convient de remarquer que le lemme 9.4 ne permet pas de montrer que la solution compensée $\overline{x}$ est aussi précise que si elle avait été calculée par substitution en précision

doublée, avec arrondi final vers la précision de travail. En d'autres termes, ce lemme ne permet pas d'obtenir une borne sur l'erreur relative $\|x-\overline{x}\|_\infty/\|\overline{x}\|_\infty$ de la même forme que l'inégalité (9.9). Rappelons ici cette borne, en notant $\widehat{x}_d$ la solution calculée en précision doublée $\mathbf{u}^2$, puis arrondie vers la précision courante $\mathbf{u}$:

$$\frac{\|\widehat{x}_d - x\|_\infty}{\|\widehat{x}\|_\infty} \leq \mathbf{u} + n\mathbf{u}^2 \operatorname{cond}(T,x) + \mathcal{O}(\mathbf{u}^3). \tag{9.17}$$

On constate que cette borne sur l'erreur relative entachant $\widehat{x}_d$ dépend essentiellement du nombre de conditionnement $\operatorname{cond}(T,x)$, tel que

$$\operatorname{cond}(T,x) = \frac{\||T^{-1}||T||x|\|_\infty}{\|x\|_\infty}.$$

Par contre, toute borne sur l'erreur relative entachant $\overline{x}$, obtenue à partir de l'inégalité (9.13), fera intervenir le facteur $\gamma_{2n}\,\mathrm{K}(T,x)$, avec par définition

$$\mathrm{K}(T,x) := \frac{\|(|T^{-1}||T|)^2|x|\|_\infty}{\|x\|_\infty}. \tag{9.18}$$

De plus, remarquons que cela reste vrai, même si le résidu est calculé plus précisément qu'en précision doublée, par exemple si $\widehat{r}$ est calculé comme étant l'arrondi au plus proche du résidu exact : dans ce cas, $|\widehat{r}-r| \leq |r|$, mais le facteur $\mathrm{K}(T,x)$ intervient toujours dans la borne d'erreur (9.13).

La meilleure majoration que nous puissions écrire pour $\mathrm{K}(T,x)$ est :

$$\mathrm{K}(T,x) \leq \||T^{-1}||T|\|_\infty \frac{\||T^{-1}||T||x|\|_\infty}{\|x\|_\infty} = \operatorname{cond}(T)\operatorname{cond}(T,x).$$

D'autre part, comme $I \leq |T^{-1}||T|$, on a $|T^{-1}||T||x| \leq (|T^{-1}||T|)^2|x|$. On a donc

$$\operatorname{cond}(T,x) \leq \mathrm{K}(T,x) \leq \operatorname{cond}(T)\operatorname{cond}(T,x).$$

Notons bien qu'il peut y avoir égalité entre les trois quantités intervenant dans la relation précédente : il suffit pour cela de supposer que T est une matrice diagonale. La majoration $\mathrm{K}(T,x) \leq \operatorname{cond}(T)\operatorname{cond}(T,x)$ peut donc être atteinte, mais semble constituer une grossière surestimation de $\mathrm{K}(T,x)$ dans le cas général, comme nous le verrons dans nos expériences numériques.

Dans les bornes *a priori* sur la précision du résultat compensé, que nous déduirons du lemme 9.4 dans les sections suivantes, nous ferrons donc toujours intervenir explicitement le terme $\mathrm{K}(T,x)$. Nous monterons, dans la section dédiée aux expériences numériques, comment interpréter ces bornes d'erreurs.

9.4 Calcul du résidu à l'aide des transformations exactes

Soit $Tx = b$ un système triangulaire d'équations linéaires, avec $T \in \mathbf{F}^{n\times n}$ une matrice triangulaire inférieure non singulière et $b \in \mathbf{F}^{n\times 1}$. Soit également $\widehat{x} = \mathsf{TRSV}(T,x)$ (algorithme 9.1) la solution approchée du système $Tx = b$ calculée par substitution. Dans cette section, nous proposons de calculer une approximation $\widehat{r}$ du résidu $r = b - Tx$ par une

méthodes basée sur l'utilisation des transformations exactes des opérations élémentaires. Le résidu approché $\widehat{r}$ est en effet déduit des erreurs d'arrondis générées par les opérations arithmétiques intervenant dans l'algorithme TRSV. Nous montrons que le résidu ainsi calculé est d'une précision similaire à celle que l'on obtiendrait en le calculant $b - T\widehat{x}$ en précision doublée $\mathbf{u}^2$.

De cette méthode de calcul du résidu, nous déduisons ensuite un algorithme de substitution compensé que nous appelons CompTRSV (algorithme 9.9). Nous formulerons une borne sur la précision de la solution compensée calculée par cet algorithme, puis nous montrerons comment implanter CompTRSV en pratique.

9.4.1 Principe de la méthode de calcul du résidu proposée

Pour facilité la lecture des énoncés qui vont suivre, nous reproduisons ci-dessous l'algorithme TRSV, en indiquant après chaque opération flottante le symbole désignant l'erreur d'arrondi qu'elle génère.

function $\widehat{x} = \mathsf{TRSV}(T, b)$
  for $k = 1 : n$
    $\widehat{s}_{k,0} = b_k$
    for $i = 1 : k-1$
      $\widehat{p}_{k,i} = t_{k,i} \otimes \widehat{x}_i$ {erreur d'arrondi $\pi_{k,i} \in \mathbf{F}$ }
      $\widehat{s}_{k,i} = \widehat{s}_{k,i-1} \ominus \widehat{p}_{k,i}$ {erreur d'arrondi $\sigma_{k,i} \in \mathbf{F}$ }
    end
    $\widehat{x}_k = \widehat{s}_{k,k-1} \oslash t_{k,k}$ {erreur d'arrondi $\rho_k/t_{k,k}$, avec $\rho_k \in \mathbf{F}$ }
  end

On considère les erreurs d'arrondis générées par les opérations arithmétiques intervenant dans l'algorithme TRSV (algorithme 9.1).

- $\pi_{k,i} \in \mathbf{F}$ est l'erreur d'arrondi entachant le produit $t_{k,i} \otimes \widehat{x}_k$.
- $\sigma_{k,i} \in \mathbf{F}$ est l'erreur d'arrondi générée par la soustraction $\widehat{s}_{k,i-1} \ominus \widehat{p}_{k,i}$.
- $\rho_k \in \mathbf{F}$, tel que $\rho_k/t_{k,k}$ est l'erreur d'arrondi entachant la division $\widehat{s}_{k-1} \oslash t_{k,k}$.

Plus formellement, on a les relations suivantes pour $k = 1 : n$ et $i = 1 : k-1$,

$$\widehat{p}_{k,i} = t_{k,i}\widehat{x}_k - \pi_{k,i}, \quad \text{avec} \quad \pi_{k,i} \in \mathbf{F} \quad \text{et} \quad |\pi_{k,i}| \leq \mathbf{u}|\widehat{p}_{k,i}|,$$

$$\widehat{s}_{k,i} = \widehat{s}_{k,i-1} - \widehat{p}_{k,i} - \sigma_{k,i}, \quad \text{avec} \quad \sigma_{k,i} \in \mathbf{F} \quad \text{et} \quad |\sigma_{k,i}| \leq \mathbf{u}|\widehat{s}_{k,i}|.$$

De plus, pour $k = 1 : n$, on a

$$\widehat{x}_k = \frac{\widehat{s}_{k,k-1} - \rho_k}{t_{k,k}}, \quad \text{avec} \quad \rho_k \in \mathbf{F} \quad \text{et} \quad |\rho_k| \leq \mathbf{u}|t_{k,k}\widehat{x}_k|.$$

Ces relations nous permettent de démontrer les propositions 9.5 et 9.7 ci-dessous.

Proposition 9.5. *Soit $\widehat{x} = (\widehat{x}_1, \ldots, \widehat{x}_n)^T \in \mathbf{F}^n$ tel que $\widehat{x} = \mathsf{TRSV}(T, b)$. Soit également $r = (r_1, \ldots, r_n)^T \in \mathbf{R}^n$ le résidu associé à la solution approchée $\widehat{x}$ du système $Tx = b$, c'est à dire $r = b - T\widehat{x}$. On a alors,*

$$r_k = \rho_k + \sum_{i=1}^{k-1} \sigma_{k,i} - \pi_{k,i}, \quad \textit{pour} \quad k = 1 : n. \tag{9.19}$$

Preuve. On fixe k entre 1 et n. Pour $i = 1 : k-1$, on a $\widehat{s}_{k,i} = \widehat{s}_{k,i-1} - t_{k,i}\widehat{x}_k + \pi_{k,i} - \sigma_{k,i}$. Comme $\widehat{s}_{k,0} = b_k$, on peut montrer par récurrence que

$$\widehat{s}_{k,k-1} = b_k - \sum_{i=1}^{k-1} t_{k,i}\widehat{x}_k + \sum_{i=1}^{k-1} \pi_{k,i} - \sigma_{k,i}.$$

De plus, $t_{k,k}\widehat{x}_k = \widehat{s}_{k,k-1} - \rho_k$, d'où

$$t_{k,k}\widehat{x}_k = b_k - \sum_{i=1}^{k-1} t_{k,i}\widehat{x}_k - \rho_k + \sum_{i=1}^{k-1} \pi_{k,i} - \sigma_{k,i}.$$

En procédant par équivalence, on obtient

$$b_k - \sum_{i=1}^{k} t_{k,i}\widehat{x}_i = \rho_k + \sum_{i=1}^{k-1} \sigma_{k,i} - \pi_{k,i}.$$

Or, par définition, $r = b - Tx$, donc $r_k = b_k - \sum_{i=1}^{k} t_{k,i}\widehat{x}_i$. On a donc

$$r_k = \rho_k + \sum_{i=1}^{k-1} \sigma_{k,i} - \pi_{k,i},$$

ce qui démontre la proposition. ■

La proposition 9.5 fournit une relation exacte, exprimant le résidu exact $r = b - T\widehat{x}$ en fonction des erreurs d'arrondi élémentaires générées par l'algorithme de substitution. On déduit de cette proposition l'algorithme TRSVResidual ci-dessous, qui calcule une approximation $\widehat{r}$ de r en même temps que la solution approchée $\widehat{x} = \mathsf{TRSV}(T, b)$ du système.

Algorithme 9.6.

```
function [x̂, r̂] = TRSVResidual(T, b)
  for k = 1 : n
    ŝ_{k,0} = b_k ; r̂_{k,0} = 0
    for i = 1 : k − 1
      [p̂_{k,i}, π_{k,i}] = TwoProd(t_{k,i}, x̂_i)
      [ŝ_{k,i}, σ_{k,i}] = TwoSum(ŝ_{k,i−1}, −p̂_{k,i})
      r̂_{k,i} = r̂_{k,i−1} ⊕ (σ_{k,i} ⊖ π_{k,i})
    end
    [x̂_k, ρ_k] = DivRem(ŝ_{k−1}, t_{k,k})
    r̂_k = ρ_k ⊕ r̂_{k,i−1}
  end
```

Il est aisé de constater que les composantes $\widehat{r}_1, \ldots, \widehat{r}_k$ du vecteur $\widehat{r}$ sont effectivement calculées par l'algorithme TRSVResidual d'après la relation (9.19). Dans la sous-section suivante, nous montrons que le résidu approché $\widehat{r}$ calculé par TRSVResidual est aussi précis que s'il avait été calculé en précision doublée.

9.4.2 Précision du résidu calculé par TRSVResidual

La proposition suivante fournit une borne sur l'erreur entachant le résidu calculé par l'algorithme TRSVResidual.

Proposition 9.7. *Avec les mêmes hypothèses que dans la proposition 9.5, l'erreur entachant le résidu calculé par l'algorithme* TRSVResidual *satisfait*

$$|\widehat{r} - r| \leq \gamma_n \gamma_{n+1}(|b| + |T||x|). \tag{9.20}$$

En faisant intervenir le conditionnement pour le calcul du résidu défini par la relation (9.10), la borne (9.20) indique que l'erreur relative entachant le résidu calculé $\widehat{r}$ sera majorée comme suit,

$$\frac{\|\widehat{r} - r\|_\infty}{\|r\|_\infty} \leq n^2 \mathbf{u}^2 \operatorname{cond}(r) + \mathcal{O}(\mathbf{u}^3). \tag{9.21}$$

Cette dernière inégalité laisse à penser que le résidu $\widehat{r}$ pourrait éventuellement être calculé avec une erreur relative inférieure à l'unité d'arrondi $\mathbf{u}$. Il convient cependant de rappeler que le calcul du résidu est un calcul mal conditionné à la précision $\mathbf{u}$: comme le montre la proposition 9.3, on a toujours $\mathbf{u} \lesssim \mathbf{u}^2 \operatorname{cond}(r)$. La proposition 9.7 ne garantit donc en aucun cas une erreur relative inférieure à l'unité d'arrondi pour $\widehat{r}$.

Néanmoins, il est important de noter que le facteur $n^2\mathbf{u}^2$ présent devant $\operatorname{cond}(r)$ dans la relation (9.21), que l'on considère comme un $\mathcal{O}(\mathbf{u}^2)$, nous indique que le résidu $\widehat{r}$ calculé par TRSVResidual est aussi précis que s'il avait été calculé en précision doublée.

La preuve de la proposition 9.7 repose sur le lemme suivant.

Lemme 9.8. *Avec les mêmes hypothèses que dans la proposition 9.5, et en utilisant les notations de l'algorithme 9.6, on a*

$$|\rho_k| + \sum_{i=1}^{k-1} |\sigma_{k,i}| + |\pi_{k,i}| \leq \gamma_{k+1} \left(|b_k| + \sum_{i=1}^{k} |t_{k,i}\widehat{x}_i| \right), \quad \textit{pour} \quad k = 1 : n.$$

Preuve. On a $|\rho_k| \leq \mathbf{u}|t_{k,k}\widehat{x}_k|$, $|\sigma_{k,i}| \leq \mathbf{u}|\widehat{s}_{k,i}|$ et $|\pi_{k,i}| \leq \mathbf{u}|t_{k,i}\widehat{x}_i|$, donc

$$|\rho_k| + \sum_{i=1}^{k-1} |\sigma_{k,i}| + |\pi_{k,i}| \leq \mathbf{u} \left(\sum_{i=1}^{k} |t_{k,i}\widehat{x}_i| + \sum_{i=1}^{k-1} |\widehat{s}_{k,i}| \right).$$

Il nous faut donc majorer les $|\widehat{s}_{k,i}|$. On a

$$\begin{array}{lcll}
\widehat{s}_{k,0} & = & b_k, & \\
\widehat{s}_{k,1} & = & \langle 1\rangle(\widehat{s}_{k,0} - \langle 1\rangle t_{k,1}\widehat{x}_1) & = \langle 1\rangle b_k - \langle 2\rangle t_{k,1}\widehat{x}_1, \\
\widehat{s}_{k,2} & = & \langle 1\rangle(\widehat{s}_{k,1} - \langle 1\rangle t_{k,2}\widehat{x}_2) & = \langle 2\rangle b_k - \langle 3\rangle t_{k,1}\widehat{x}_1 - \langle 2\rangle t_{k,2}\widehat{x}_2, \\
\vdots & & \vdots & \\
\widehat{s}_{k,i} & = & \langle 1\rangle(\widehat{s}_{k,i-1} - \langle 1\rangle t_{k,i}\widehat{x}_i) & = \langle i\rangle b_k - \langle i+1\rangle t_{k,1}\widehat{x}_1 - \langle i\rangle t_{k,2}\widehat{x}_2 - \cdots - \langle 2\rangle t_{k,i}\widehat{x}_i.
\end{array}$$

Comme chacun des compteurs d'erreur $\langle l\rangle$ est majoré en valeur absolue par $1 + \gamma_l$, on obtient

$$|\widehat{s}_{k,i}| \leq (1 + \gamma_{i+1}) \left(|b_k| + \sum_{j=1}^{i} |t_{k,j}\widehat{x}_j| \right), \quad \text{pour} \quad i = 1 : k-1.$$

On a alors

$$\begin{aligned}
|\rho_k| + \sum_{i=1}^{k-1} |\sigma_{k,i}| + |\pi_{k,i}| &\leq \mathbf{u}\left(\sum_{i=1}^{k} |t_{k,i}\widehat{x}_i| + \sum_{i=1}^{k-1}(1+\gamma_{i+1})\left(|b_k| + \sum_{j=1}^{i} |t_{k,j}\widehat{x}_j|\right)\right) \\
&\leq \mathbf{u}\left(\sum_{i=1}^{k} |t_{k,i}\widehat{x}_i| + (k-1)(1+\gamma_k)\left(|b_k| + \sum_{i=1}^{k-1} |t_{k,i}\widehat{x}_i|\right)\right) \\
&\leq k\mathbf{u}(1+\gamma_k)\left(|b_k| + \sum_{i=1}^{k} |t_{k,i}\widehat{x}_i|\right).
\end{aligned}$$

Comme $k\mathbf{u}(1+\gamma_k) \leq \gamma_{k+1}$, cela achève la preuve du lemme. ■

Preuve de la proposition 9.7. Soit $k = 1 : n$ fixé. Commençons par chercher une borne sur $|\widehat{r}_k - r_k|$ faisant intervenir $|\rho_k|$ ainsi que les $|\sigma_{k,i}|$ et $|\pi_{k,i}|$. On a,

$$\begin{array}{lcll}
\widehat{r}_{k,0} & = & 0 & \\
\widehat{r}_{k,1} & = & \langle 1\rangle(\sigma_{k,1} - \pi_{k,1}) & \\
\widehat{r}_{k,2} & = & \langle 1\rangle(\widehat{r}_{k,1} + \langle 1\rangle(\sigma_{k,2} - \pi_{k,2})) & = \quad \langle 2\rangle(\sigma_{k,1} - \pi_{k,1}) + \langle 2\rangle(\sigma_{k,2} - \pi_{k,2}) \\
\widehat{r}_{k,3} & = & \langle 1\rangle(\widehat{r}_{k,2} + \langle 1\rangle(\sigma_{k,3} - \pi_{k,3})) & = \quad \langle 3\rangle(\sigma_{k,1} - \pi_{k,1}) + \langle 3\rangle(\sigma_{k,2} - \pi_{k,2}) + \langle 2\rangle(\sigma_{k,3} - \pi_{k,3}).
\end{array}$$

On peut alors montrer par induction que

$$\widehat{r}_{k,k-1} = \langle k-1\rangle(\sigma_{k,1} - \pi_{k,1}) + \langle k-1\rangle(\sigma_{k,2} - \pi_{k,2}) + \langle k-2\rangle(\sigma_{k,3} - \pi_{k,3}) + \cdots \\ \cdots + \langle 2\rangle(\sigma_{k,k-1} - \pi_{k,k-1}).$$

Comme $\widehat{r}_k = \langle 1\rangle(\rho_k + \widehat{r}_{k,k-1})$, on obtient

$$\widehat{r}_k = \langle 1\rangle\rho_k + \langle k\rangle(\sigma_{k,1} - \pi_{k,1}) + \langle k\rangle(\sigma_{k,2} - \pi_{k,2}) + \langle k-1\rangle(\sigma_{k,3} - \pi_{k,3}) + \cdots \\ \cdots + \langle 3\rangle(\sigma_{k,k-1} - \pi_{k,k-1}).$$

Puisque $r_k = \rho_k + \sum_{i=1}^{k-1} \sigma_{k,i} - \pi_{k,i}$, on en déduit alors que

$$|\widehat{r}_k - r_k| \leq \gamma_k\left(|\rho_k| + \sum_{i=1}^{k-1} |\sigma_{k,i}| + |\pi_{k,i}|\right).$$

En utilisant le lemme précédent, on obtient

$$|\widehat{r}_k - r_k| \leq \gamma_k\gamma_{k+1}\left(|b_k| + \sum_{i=1}^{k} |t_{k,i}\widehat{x}_i|\right),$$

ce qui démontre le résultat annoncé. ■

9.4.3 Algorithme CompTRSV : précision du résultat compensé

On introduit maintenant l'algorithme CompTRSV qui formalise la méthode de résolution compensée décrite en début de chapitre. Notre but est ici de borner l'erreur entachant la solution compensée $\overline{x}$.

Algorithme 9.9.

function $\overline{x} = \mathsf{CompTRSV}(T, b)$
  $[\widehat{x}, \widehat{r}] = \mathsf{TRSVResidual}(T, b)$
  $\widehat{c} = \mathsf{TRSV}(T, \widehat{r})$
  $\overline{x} = \widehat{x} \oplus \widehat{c}$

Théorème 9.10. *On considère le système triangulaire $Tx = b$ avec $T \in \mathbf{F}^{n\times n}$ non singulière telle que $\gamma_n \operatorname{cond}(T) < 1$, et $b \in \mathbf{F}^{n\times 1}$. Soit $\overline{x} = \mathsf{CompTRSV}(T, x)$, la solution compensée de ce système triangulaire calculée par l'algorithme 9.9. Alors l'erreur directe entachant $\overline{x}$ satisfait*

$$|\overline{x} - x| \leq \mathbf{u}|x| + (2\gamma_n\gamma_{n+1} + \gamma_n^2)(|T^{-1}||T|)^2|x| + \mathcal{O}(\mathbf{u}^3).$$

Interprétons le théorème 9.10 en terme d'erreur relative, en faisant intervenir le facteur $\mathrm{K}(T, x)$, défini par la relation (9.18) :

$$\frac{\|\overline{x} - x\|_\infty}{\|x\|_\infty} \leq \mathbf{u} + n(3n+2)\mathbf{u}^2\,\mathrm{K}(T, x) + \mathcal{O}(\mathbf{u}^3). \tag{9.22}$$

La borne d'erreur (9.22) ne permet pas de garantir que l'algorithme CompTRSV est aussi précis que si la solution compensée $\widehat{x}$ avait été calculée en précision doublée, avec arrondi final vers la précision courante : comme nous l'avons déjà indiqué à la Section 9.3, la borne d'erreur (9.22) fait en effet intervenir la facteur $\mathrm{K}(T, x)$ qui n'intervient pas dans la borne d'erreur (9.17).

Les expériences numériques que nous rapportons à la Section 9.6 indiquent néanmoins que la solution compensée calculée par CompTRSV semble en pratique être aussi précise que si elle avait été calculée en précision doublée.

Pour la preuve du théorème 9.10, nous démontrons au préalable le lemme suivant, qui fournit une borne sur l'erreur directe entachant le terme correctif calculé $\widehat{c}$.

Preuve du théorème 9.10. D'après la proposition 9.7, l'erreur directe entachant $\widehat{r}$ est majorée comme suit,

$$|\widehat{r} - r| \leq \gamma_n\gamma_{n+1}(|b| + |T||\widehat{x}|).$$

Comme $|\widehat{x} - x| \leq |F||x|$, on a $|\widehat{x}| \leq |x| + |\widehat{x} - x| \leq (I + |F|)|x|$. Ainsi, on a

$$\begin{aligned} |\widehat{r} - r| &\leq \gamma_n\gamma_{n+1}(|b| + |T|(I + |F|)|x|) \\ &\leq \gamma_n\gamma_{n+1}(|T| + |T|(I + |F|))|x| \\ &\leq \gamma_n\gamma_{n+1}|T|(2I + |F|)|x|. \end{aligned}$$

Comme $|F| \leq \gamma_n|T^{-1}||T| + \mathcal{O}(\mathbf{u}^2)$, on en déduit que

$$|\widehat{r} - r| \leq 2\gamma_n\gamma_{n+1}|T||x| + \mathcal{O}(\mathbf{u}^3).$$

Notons que le membre droit de l'inégalité précédente est en $\mathcal{O}(\mathbf{u}^2)$, donc $|\widehat{r} - r| \leq \mathbf{u}|r| + \mathcal{O}(\mathbf{u}^2)$. Ainsi, d'après le lemme 9.4, on a

$$|\overline{x} - x| \leq \mathbf{u}|x| + 2\gamma_n\gamma_{n+1}|T^{-1}||T||x| + \gamma_n^2(|T^{-1}||T|)^2|x| + \mathcal{O}(\mathbf{u}^3). \tag{9.23}$$

Comme de plus $|T^{-1}||T||x| \leq (|T^{-1}||T|)^2|x|$, on en déduit le résultat annoncé. ■

9.4.4 Implantation pratique de CompTRSV

Telle que décrite par l'algorithme 9.9, notre méthode compensée de résolution des systèmes triangulaires nécessite de parcourir deux fois les éléments de la matrice T : une première fois pour calculer la solution initiale $\widehat{x}$ et le résidu $\widehat{r}$, une seconde pour calculer le terme correctif $\widehat{c}$. Cela peut être facilement évité, en combinant les algorithmes 9.9 et 9.6, comme nous le montrons avec l'algorithme 9.11 ci-dessous.

Algorithme 9.11.

function $\overline{x} = \mathsf{CompTRSV}(T, b)$
  for $k = 1 : n$
    $\widehat{s}_{k,0} = b_k$; $\widehat{q}_{k,0} = \widehat{c}_{k,0} = 0$
    for $i = 1 : k-1$
      $[\widehat{p}_{k,i}, \pi_{k,i}] = \mathsf{TwoProd}(t_{k,i}, \overline{x}_i)$
      $[\widehat{s}_{k,i}, \sigma_{k,i}] = \mathsf{TwoSum}(\widehat{s}_{k,i-1}, -\widehat{p}_{k,i})$
      $\widehat{q}_{k,i} = \widehat{q}_{k,i-1} \oplus \widehat{c}_i \otimes \widehat{t}_{k,i}$
      $\widehat{r}_{k,i} = \widehat{r}_{k,i-1} \oplus (\sigma_{k,i} \ominus \pi_{k,i})$
    end
    $[\widehat{x}_k, \rho_k] = \mathsf{DivRem}(\widehat{s}_{k-1}, t_{k,k})$
    $\widehat{r}_k = \rho_k \oplus \widehat{r}_{k,i-1}$
    $\widehat{c}_k = (r_k \ominus \widehat{q}_{k,k-1}) \oslash t_{k,k}$
  end
  $\overline{x}_1 = \widehat{x}_1$
  for $k = 2 : n$
    $\overline{x}_k = \widehat{x}_k \oplus \widehat{c}_k$
  end

Il est à noter qu'il n'est pas nécessaire d'effectuer la correction $\overline{x}_1 = \widehat{x}_1 \oplus \widehat{y}_1$. En effet, T est triangulaire inférieure, donc $\widehat{x}_1 = b_1 \oslash t_{1,1}$: en arrondi au plus proche, $\widehat{x}_1$ est déjà la meilleure approximation possible de x_1 en arithmétique flottante. On fixe donc simplement $\overline{x}_1 = \widehat{x}_1$ dans l'algorithme 9.11.

Notons finalement que l'algorithme CompTRSV (algorithme 9.11) nécessite $27n^2/2 + 21n/2 + \mathcal{O}(1)$ opérations flottantes.

9.5 Calcul du résidu en précision doublée avec Dot2

Soit $Tx = b$ un système triangulaire d'équations linéaires, avec $T \in \mathbf{F}^{n\times n}$ une matrice triangulaire inférieure non singulière et $b \in \mathbf{F}^{n\times 1}$. Soit également $\widehat{x} = \mathsf{TRSV}(T, b)$ la solution approchée du système $Tx = b$ calculée par substitution. On s'intéresse ici au calcul du résidu à l'aide de l'algorithme Dot2 [70] (algorithme 3.23). Pour formuler clairement ce calcul, définissons

- $(b_k, t_{k,1:k})$ le vecteur $(b_k, t_{k,1}, \ldots, t_{k,k}) \in \mathbf{F}^{k+1}$,
- et $(1; -\widehat{x}_{1:k})$ le vecteur $(1, -\widehat{x}_1, \ldots, -\widehat{x}_k)^T \in \mathbf{F}^{k+1}$.

Alors l'élément r_k du résidu $r = b - T\widehat{x}$ est approché par

$$\widehat{r}_k = \mathsf{Dot2}\big((b_k, t_{k,1:k}), (1; -\widehat{x}_{1:k})\big). \tag{9.24}$$

L'approche classique pour le calcul du résidu à l'aide de Dot2 serait de calculer d'abord $\widehat{x} = \mathsf{TRSV}(T, b)$, puis de calculer successivement les composantes de $\widehat{r}$ conformément à la relation précédente. Dans la suite de cette section, nous montrons qu'il est possible de combiner ces deux calculs : nous formulerons ainsi l'algorithme TRSVResidual2, qui calcule à la fois $\widehat{x} = \mathsf{TRSV}(T, b)$ et $\widehat{r}$ d'après la relation (9.24).

Nous montrons que TRSVResidual2 est en réalité très proche de TRSVResidual vu à la section précédente : entre ces deux algorithmes, seul change l'ordre de quelques opérations flottantes. Comme nous le verrons, cela modifie naturellement la valeur du résidu calculé, mais ne peut en aucun cas en changer grandement la précision.

Par soucis d'exhaustivité, nous formulerons ensuite l'algorithme CompTRSV2, qui calcule une solution compensée $\overline{x}$ du système $Tx = b$, à l'aide de TRSVResidual2. Nous montrerons que la solution compensée calculée par cette méthode satisfait naturellement une borne d'erreur similaire à celle vérifiée par l'algorithme TRSVResidual.

Dans la mesure où les algorithmes CompTRSV et CompTRSV2sont extrêmement semblables, on doit s'attendre en pratique à ce qu'ils présentent le même comportement numérique. Dans les sections suivantes, notamment dans nos expériences numériques, nous nous concentrerons donc sur l'étude de CompTRSV, et ne nous intéresserons plus à CompTRSV2.

9.5.1 Lien entre TRSVResidual2 et TRSVResidual

On suppose que la solution approchée $\widehat{x} = (x_1, \ldots, x_n)^T = \mathsf{TRSV}(T, b)$ est déjà calculée. On considère la portion de code ci-dessous, dans laquelle l'algorithme Dot2(algorithme 3.23) a été *inliné*, sans changement de l'ordre des opérations, afin de calculer les composantes de $\widehat{r}$ conformément à la relation (9.24).

for $k = 1 : n$
  $\widehat{s}_{k,0} = b_k$; $\widehat{r}_{k,0} = 0$
  for $i = 1 : k$
    $[\widehat{p}_{k,i}, \pi_{k,i}] = \mathsf{TwoProd}(t_{k,i}, \widehat{x}_i)$
    $[\widehat{s}_{k,i}, \sigma_{k,i}] = \mathsf{TwoSum}(\widehat{s}_{k,i-1}, -\widehat{p}_{k,i})$
    $\widehat{r}_{k,i} = \widehat{r}_{k,i-1} \oplus (\sigma_{k,i} \ominus \pi_{k,i})$
  end
  $\widehat{r}_k = \widehat{s}_{k,k} \oplus \widehat{r}_{k,k}$
end

Dans l'algorithme ci-dessus, il n'est pas difficile de constater que $\widehat{x}_k = \widehat{s}_{k,k-1} \oslash t_{k,k}$. En effet, puisque $[\widehat{p}_{k,i}, \pi_{k,i}] = \mathsf{TwoProd}(t_{k,i}, \widehat{x}_i)$ et $[\widehat{s}_{k,i}, \sigma_{k,i}] = \mathsf{TwoSum}(\widehat{s}_{k,i-1}, -\widehat{p}_{k,i})$, on a en particulier $\widehat{p}_{k,i} = t_{k,i} \otimes \widehat{x}_i$ et $\widehat{s}_{k,i} = \widehat{s}_{k,i-1} \ominus \widehat{p}_{k,i}$. L'algorithme précédent effectue donc une partie des opérations nécessaires au calcul de $\widehat{x} = \mathsf{TRSV}(T, b)$ par substitution. Pour éviter ces calculs redondants, nous le modifions de manière à ce qu'il calcule à la fois $\widehat{x}$ et $\widehat{r}$. On a ainsi,

for $k = 1 : n$
  $\widehat{s}_{k,0} = b_k$; $\widehat{r}_{k,0} = 0$
  for $i = 1 : k - 1$
    $[\widehat{p}_{k,i}, \pi_{k,i}] = \mathsf{TwoProd}(t_{k,i}, \widehat{x}_i)$
    $[\widehat{s}_{k,i}, \sigma_{k,i}] = \mathsf{TwoSum}(\widehat{s}_{k,i-1}, -\widehat{p}_{k,i})$

$\widehat{r}_{k,i} = \widehat{r}_{k,i-1} \oplus (\sigma_{k,i} \ominus \pi_{k,i})$
end
$\widehat{x}_k = \widehat{s}_{k,k-1} \oslash t_{k,k}$
$[\widehat{p}_{k,k}, \pi_{k,k}] = \mathsf{TwoProd}(t_{k,k}, \widehat{x}_k)$
$[\widehat{s}_{k,k}, \sigma_{k,k}] = \mathsf{TwoSum}(\widehat{s}_{k,k-1}, -\widehat{p}_{k,k})$
$\widehat{r}_{k,k} = \widehat{r}_{k,k-1} \oplus (\sigma_{k,k} \ominus \pi_{k,k})$
$\widehat{r}_k = \widehat{s}_{k,k} \oplus \widehat{r}_{k,k}$
end

Montrons de plus que dans l'algorithme ci-dessus, on a $\sigma_{k,k} = 0$, ce qui revient à montrer que la soustraction $\widehat{s}_{k,k} = \widehat{s}_{k,k-1} \ominus \widehat{p}_{k,k}$ est une opération exacte. Pour ce faire, définissons ρ_k tel que

$$\frac{\widehat{s}_{k,k-1}}{t_{k,k}} = \widehat{x}_k + \frac{\rho_k}{t_{k,k}}. \tag{9.25}$$

Comme $\widehat{x}_k = \widehat{s}_{k,k-1} \oslash t_{k,k}$ et $[\widehat{p}_{k,k}, \pi_{k,k}] = \mathsf{TwoProd}(t_{k,k}, \widehat{x}_k)$, d'après le théorème 3.14 portant sur l'algorithme DivRem (algorithme 3.13), on sait que

$$\rho_k = (\widehat{s}_{k,k-1} \ominus \widehat{p}_{k,k}) \ominus \pi_{k,k} \in \mathbf{F}.$$

D'autre part, comme $[\widehat{p}_{k,k}, \pi_{k,k}] = \mathsf{TwoProd}(t_{k,k}, \widehat{x}_k)$, et en utilisant (9.25), on a

$$\begin{aligned} \widehat{s}_{k,k-1} - \widehat{p}_{k,k} &= \widehat{s}_{k,k-1} - (t_{k,k}\widehat{x}_k - \pi_{k,k}) \\ &= \widehat{s}_{k,k-1} - (\widehat{s}_{k,k-1} - \rho_k - \pi_{k,k}) \\ &= \rho_k + \pi_{k,k}. \end{aligned}$$

Donc $\widehat{s}_{k,k-1} - \widehat{p}_{k,k} = \widehat{s}_{k,k-1} \ominus \widehat{p}_{k,k}$, ce qui signifie que cette opération n'est entachée d'aucune erreur d'arrondi, d'où $\sigma_{k,k} = 0$. On formule donc l'algorithme TRSVResidual2de la façon suivante.

Algorithme 9.12.

function $[\widehat{x}, \widehat{r}] = \mathsf{TRSVResidual2}(T, b)$
for $k = 1 : n$
$\widehat{s}_{k,0} = b_k$; $\widehat{r}_{k,0} = 0$
for $i = 1 : k-1$
$[\widehat{p}_{k,i}, \pi_{k,i}] = \mathsf{TwoProd}(t_{k,i}, \widehat{x}_i)$
$[\widehat{s}_{k,i}, \sigma_{k,i}] = \mathsf{TwoSum}(\widehat{s}_{k,i-1}, -\widehat{p}_{k,i})$
$\widehat{r}_{k,i} = \widehat{r}_{k,i-1} \oplus (\sigma_{k,i} \ominus \pi_{k,i})$
end
$\widehat{x}_k = \widehat{s}_{k,k-1} \oslash t_{k,k}$
$[\widehat{p}_{k,k}, \pi_{k,k}] = \mathsf{TwoProd}(t_{k,k}, \widehat{x}_k)$
$\widehat{r}_k = (\widehat{s}_{k,k-1} \ominus \widehat{p}_{k,k}) \oplus (\widehat{r}_{k,k-1} \ominus \pi_{k,k})$
end

On peut d'ors et déjà remarquer la ressemblance algorithmique entre TRSVResidual (algorithme 9.6) et TRSVResidual2 (algorithme 9.12), bien que les deux algorithmes aient été obtenues par des méthodes différentes. Afin de bien mettre en évidence cette similarité, écrivons côte à côte ces deux algorithmes. Dans l'algorithme, TRSVResidual, nous avons *inliné* les instructions de la transformation exacte DivRem (algorithme 3.13).

function $[\widehat{x},\widehat{r}] = \mathsf{TRSVResidual}(T,b)$
  for $k = 1:n$
    $\widehat{s}_{k,0} = b_k$; $\widehat{r}_{k,0} = 0$
    for $i = 1:k-1$
      $[\widehat{p}_{k,i}, \pi_{k,i}] = \mathsf{TwoProd}(t_{k,i}, \widehat{x}_i)$
      $[\widehat{s}_{k,i}, \sigma_{k,i}] = \mathsf{TwoSum}(\widehat{s}_{k,i-1}, -\widehat{p}_{k,i})$
      $\widehat{r}_{k,i} = \widehat{r}_{k,i-1} \oplus (\sigma_{k,i} \ominus \pi_{k,i})$
    end
    $\widehat{x}_k = \widehat{s}_{k,k-1} \oslash t_{k,k}$
    $[\widehat{p}_{k,k}, \pi_{k,k}] = \mathsf{TwoProd}(t_{k,k}, \widehat{x}_k)$
    $\rho_k = (\widehat{s}_{k,k-1} \ominus \widehat{p}_{k,k}) \ominus \pi_{k,k}$
    $\widehat{r}_k = \rho_k \oplus \widehat{r}_{k,k-1}$
  end

function $[\widehat{x},\widehat{r}] = \mathsf{TRSVResidual2}(T,b)$
  for $k = 1:n$
    $\widehat{s}_{k,0} = b_k$; $\widehat{r}_{k,0} = 0$
    for $i = 1:k-1$
      $[\widehat{p}_{k,i}, \pi_{k,i}] = \mathsf{TwoProd}(t_{k,i}, \widehat{x}_i)$
      $[\widehat{s}_{k,i}, \sigma_{k,i}] = \mathsf{TwoSum}(\widehat{s}_{k,i-1}, -\widehat{p}_{k,i})$
      $\widehat{r}_{k,i} = \widehat{r}_{k,i-1} \oplus (\sigma_{k,i} \ominus \pi_{k,i})$
    end
    $\widehat{x}_k = \widehat{s}_{k,k-1} \oslash t_{k,k}$
    $[\widehat{p}_{k,k}, \pi_{k,k}] = \mathsf{TwoProd}(t_{k,k}, \widehat{x}_k)$
    $\widehat{r}_k = (\widehat{s}_{k,k-1} \ominus \widehat{p}_{k,k}) \oplus (\widehat{r}_{k,k-1} \ominus \pi_{k,k})$
  end

On constate ainsi clairement qu'entre les algorithmes TRSVResidual et TRSVResidual2, seul change l'ordre des opérations effectuées lors du calculs de $\widehat{r}_k$ en fonction de $\widehat{s}_{k,k-1}$, $\widehat{p}_{k,k}$, $\widehat{r}_{k,k-1}$ et $\pi_{k,k}$. Du fait de ce changement dans l'ordre de ces opérations, les résidus approchés calculés par ces deux algorithmes seront en généralement différents. Mais on ne pourra observer en pratique aucune différence significative entre la précision du résidu calculé par TRSVResidual celle du résidu calculé par TRSVResidual2.

Proposition 9.13. *Soit $Tx = b$ un système triangulaire d'équations linéaires, avec $T \in \mathbf{F}^{n\times n}$ une matrice triangulaire inférieure non singulière et $b \in \mathbf{F}^{n\times 1}$. Soient $\widehat{x},\widehat{r} \in \mathbf{F}^n$ tels que $[\widehat{x},\widehat{r}] = \mathsf{TRSVResidual2}(T,b)$ (algorithme 9.12). Alors $\widehat{x} = \mathsf{TRSV}(T,b)$ (algorithme 9.1) et l'erreur directe entachant le résidu calculé $\widehat{r}$ est majorée comme suit,*

$$|\widehat{r} - r| \leq \mathbf{u}|r| + \gamma_{n+1}^2(|b| + |T||x|).$$

Preuve. Pour $k = 1:n$, on a $\widehat{r}_k = \mathsf{Dot2}\big((b_k, t_{k,1:k}), (1; -\widehat{x}_{1:k})\big)$. D'après le théorème 3.24, l'erreur directe entachant $\widehat{r}_k$ est alors majorée comme suit,

$$|\widehat{r}_k - r_k| \leq \mathbf{u}|r_k| + \gamma_{k+1}^2 \left(|b_k| + \sum_{i=1}^{k} |t_{k,i}\widehat{x}_i| \right),$$

d'où le résultat annoncé. ∎

En faisant intervenir le nombre de conditionnement pour le calcul du résidu, la précision du résidu calculé par TRSVResidual2 est majorée comme suit,

$$\frac{\|\widehat{r} - r\|_\infty}{\|r\|_\infty} \leq \mathbf{u} + n^2\mathbf{u}^2 \operatorname{cond}(r) + \mathcal{O}(\mathbf{u}^3). \tag{9.26}$$

Rappelons que le calcul du résidu est un problème mal conditionné à la précision $\mathbf{u}$: dans l'inégalité précédente, le premier terme est donc toujours négligeable devant le second. La borne obtenue en supposant que le résidu est calculé en précision doublée est donc également très proche de la borne d'erreur relative (9.21) pour l'algorithme TRSVResidual (algorithme 9.6).

9.5.2 Précision du résultat compensé calculée par CompTRSV2

On considère ci-dessous l'algorithme CompTRSV2, dans lequel la solution initiale $\widehat{x}$ et le résidu approché $\widehat{r}$ sont calculés à l'aide de TRSVResidual2. Nous fournissons ensuite une borne *a priori* sur l'erreur entachant le résultat compensé $\overline{r}$.

Algorithme 9.14.

function $\overline{x} = \mathsf{CompTRSV2}(T, b)$
  $[\widehat{x}, \widehat{r}] = \mathsf{TRSVResidual2}(T, b)$
  $\widehat{c} = \mathsf{TRSV}(T, \widehat{r})$
  $\overline{x} = \widehat{x} \oplus \widehat{c}$

Dans la mesure où les algorithmes TRSVResidual et TRSVResidual2 sont quasiment équivalents, aussi bien en ce qui concerne les opérations flottantes effectuées, que la précision du résidu calculé, on doit naturellement s'attendre à ce que CompTRSV2 (algorithme 9.14) admette une borne d'erreur similaire à celle de CompTRSV (algorithme 9.14). Nous confirmons cela à l'aide du théorème suivant.

Théorème 9.15. *On considère le système triangulaire $Tx = b$ avec $T \in \mathbf{F}^{n\times n}$ non singulière telle que $\gamma_n \operatorname{cond}(T) < 1$, et $b \in \mathbf{F}^{n\times 1}$. Soit $\overline{x} = \mathsf{CompTRSV}(T, x)$ la solution compensée de ce système triangulaire calculée par l'algorithme 9.9. Alors l'erreur directe entachant la solution compensée satisfait*

$$|\overline{x} - x| \leq \mathbf{u}|x| + (\mathbf{u}\gamma_n + 2\gamma_{n+1}^2 + \gamma_n^2)(|T^{-1}||T|)^2 + \mathcal{O}(\mathbf{u}^3).$$

Comme à l'accoutumée, interprétons la borne d'erreur précédente en termes d'erreur relative,

$$\frac{\|\overline{x} - x\|_\infty}{\|x\|_\infty} \leq \mathbf{u} + (6n^2 + 5n + 2)\mathbf{u}^2 \,\mathrm{K}(T, x) + \mathcal{O}(\mathbf{u}^3). \tag{9.27}$$

Cette borne d'erreur relative est similaire à celle obtenue pour l'algorithme CompTRSV (algorithme 9.9) présentée à la section précédente. Les mêmes conclusions s'appliquent donc. En particulier, on ne peut pas garantir que CompTRSV2 calcule une solution aussi précise que si elle avait été calculée par substitution en précision doublée.

Preuve du théorème 9.15. D'après la proposition 9.7, l'erreur directe entachant $\widehat{r}$ est majorée comme suit,

$$|\widehat{r} - r| \leq \mathbf{u}|r| + \gamma_{n+1}^2(|b| + |T||\widehat{x}|).$$

Comme $|\widehat{x} - x| \leq |F||x|$, on a $|\widehat{x}| \leq |x| + |\widehat{x} - x| \leq (I + |F|)|x|$. De plus,

$$|r| = |b - T\widehat{x}| = |(T + \Delta T)\widehat{x} - T\widehat{x}| = |\Delta T\widehat{x}| \leq |\Delta T|(I + |F|)|x|.$$

On a ainsi

$$\begin{aligned} |\widehat{r} - r| &\leq \mathbf{u}|\Delta T|(I + |F|)|x| + \gamma_{n+1}^2\big(|b| + |T|(I + |F|)|x|\big) \\ &\leq \mathbf{u}|\Delta T|(I + |F|)|x| + \gamma_{n+1}^2|T|(2I + |F|)|x|). \end{aligned}$$

Comme $|\Delta T| \leq \gamma_n|T|$ et $|F| \leq \gamma_n|T^{-1}||T| + \mathcal{O}(\mathbf{u}^2)$, on a donc

$$|r - \widehat{r}| \leq (\mathbf{u}\gamma_n + 2\gamma_{n+1}^2)|T||x| + \mathcal{O}(\mathbf{u}^3).$$

Remarquons que l'on a bien $|\widehat{r} - r| \leq \mathbf{u}|r| + \mathcal{O}(\mathbf{u}^2)$. D'après le lemme 9.4, on obtient

$$|\overline{x} - x| \leq \mathbf{u}|x| + (\mathbf{u}\gamma_n + 2\gamma_{n+1}^2)|T^{-1}||T||x| + \gamma_n^2(|T^{-1}||T|)^2 + \mathcal{O}(\mathbf{u}^3).$$

Comme de plus $|T^{-1}||T||x| \leq (|T^{-1}||T|)^2|x|$, on en déduit le résultat annoncé. ■

9.6 Expériences numériques

Comme à l'accoutumée, nos expériences numériques sont effectuées sous Matlab, en double précision IEEE-754. Ces expériences visent à mettre en évidence le comportement numérique de l'algorithme CompTRSV (algorithme 9.9). Nous montrons en particulier que la précision de la solution compensée calculée par CompTRSV semble en pratique être aussi précise que si elle avait été calculée en précision doublée. Nous proposons également une interprétations pratique de la borne (9.22) sur l'erreur relative entachant la solution compensée $\overline{x}$ calculée par CompTRSV.

9.6.1 Génération des jeux de tests

Comme indiqué en Section 9.2, la précision de la solution approchée d'un système triangulaire $Tx = b$ dépend essentiellement du conditionnement de Skeel $\mathrm{cond}(T, x)$, défini par la relation (9.6). Pour illustrer le comportement numérique des algorithmes décrits dans ce chapitre, nous avons généré des systèmes triangulaires dont le conditionnement de Skeel varie entre 10^5 et 10^{35}. Dans chacun de ces systèmes, les coefficients de la matrice T, ainsi que ceux du second membre b sont bien entendu exacts : ce sont des flottants en double précision IEEE-754.

Il convient de rappeler que $\mathrm{cond}(T, x)$ peut être arbitrairement inférieur à $\mathrm{cond}(T)$. En effet, on a

$$\mathrm{cond}(T, x) = \frac{\||T^{-1}||T||x|\|_\infty}{\|x\|_\infty} \leq \||T^{-1}||T|\|_\infty = \mathrm{cond}(T, e) = \mathrm{cond}(T).$$

Insistons donc sur le fait qu'il ne suffit pas, pour obtenir un système triangulaire $Tx = b$ mal conditionné, de générer une matrice T mal conditionnée au sens de $\mathrm{cond}(T)$. Encore faut-il choisir convenablement le second membre b, de manière à ce que le système obtenu soit également mal conditionné au sens de $\mathrm{cond}(T, x)$. Nous décrivons ici deux méthodes que nous avons utilisées.

Méthode (I) – La première méthode consiste à produire une matrice triangulaire T dont les coefficients sont générés aléatoirement dans l'intervalle $[10^{-3}, 10^3]$, selon une distribution uniforme. En générant ainsi des matrices triangulaires T de dimension $n = 40$, comme ce sera le cas par la suite, on observe que $\mathrm{cond}(T)$ varie entre 10^8 et 10^{24} environ.

En notant $e = (1, \ldots, 1)^T \in \mathbf{F}^n$ le vecteur unité, le second membre b est calculé comme étant le produit Te, calculé avec une grande précision, puis arrondi vers la double précision IEEE-754. Pour cela, le produit Te est calculé à l'aide de la bibliothèque MPFR [31] avec une précision de 265 bits[1], puis arrondi vers la double précision. Même si l'on est alors quasiment assuré que les éléments du second membre b sont des arrondis fidèles du produit exact Te, il convient d'insister sur le fait que la solution exacte du système triangulaire $Tx = b$ obtenu n'est pas, en général, le vecteur unité e.

Sur le graphique de la figure 9.1, on reporte $\mathrm{cond}(T, x)$ en fonction de $\mathrm{cond}(T)$, pour 1000 systèmes triangulaires de dimension $n = 40$ générés à l'aide de cette méthode (I). Ces 1000 systèmes triangulaires seront utilisés tout au long de ces expériences numériques. Indiquons que $\mathrm{cond}(T)$ et $\mathrm{cond}(T, x)$ sont calculés via leurs définitions respectives, après

[1] 265 bits = 5×53 bits = 5 fois la double précision IEEE-754.

avoir calculé un inverse approché de T avec une précision de 265 bits à l'aide de la bibliothèque MPFR. On constate bien sur cette figure que $\mathrm{cond}(T,x)$ est toujours nettement inférieur à $\mathrm{cond}(T)$. De plus, on constate expérimentalement que lorsque la dimension des systèmes générés est fixée à $n = 40$, il est rare de générer par cette méthode des systèmes dont le conditionnement de Skeel excède 10^{18}.

En pratique, on retiendra donc que la méthode (I) nous a permis d'obtenir des systèmes triangulaires de dimension $n = 40$ dont le conditionnement de Skeel $\mathrm{cond}(T,x)$ varie entre 10^6 et 10^{18} environ.

Méthode (II) – Le but est ici de générer des systèmes triangulaires dont le conditionnement atteint 10^{35} environ, ce qui n'est pas réalisable à l'aide de la méthode (I). La méthode que nous utilisons ici reste particulièrement empirique, et nous ne sommes pas parvenu à en décrire formellement le principe. Plutôt que d'entrer dans des explications nécessairement imprécises, nous fournissons ci-dessous le code de la fonction Matlab utilisée. Notons que nous générons ici des systèmes triangulaires inférieurs.

Pour nos expériences, nous avons générés 1000 systèmes triangulaires de dimension $n = 40$ à l'aide de la fonction `illcontri` listée ci-dessous. Pour chaque système généré, la paramètre d'entrée `c` est choisi aléatoirement dans l'intervalle $[0, 20]$: la manière dont est choisit ce paramètre a été déterminée expérimentalement de manière à générer des systèmes dont le conditionnement de Skeel varie entre 10^5 et 10^{35}. La fonction `illcondtri` fait appel à la transformation exacte VecSum (algorithme 3.18), dont le principe a été rappelé au chapitre 3.

```
function [T,b] = illcondtri(n, c)
  T = zeros(n); b = zeros(n,1);
  % Si n est impair, on se ramène au cas pair.
  if ~mod(n,2)
    T(n,n) = 1.0; b(n,1) = 1.0; n = n-1;
  end
  p = (n+1)/2; % n = 2*p-1

  % On commence par générer T(1:p, 1:n) et b(1:p)
    D = diag((ones(1,p)-2*rand(1,p)) .* linspace(10^(-c),10^(c),p));
    U = triu((1-2*rand(p)) .* (10.^round(c*(1-2*rand(p)))), 1);
    T(1:p,1:p) =  D + U;
    % A l'aide de l'algorithme VecSum, on calcule T(1:p, p+1:n) et b(1:p)
    % de manière à ce que l'on aie exactement sum(T(i,:)) = b(i).
    for i=1:p
      t = VecSum(T(i,1:p));
      T(i,p+1:2*p-1) = -t(1:p-1);
      b(i) = t(p);
    end

  % On génére maintenant T(p+1:n,p+1:n) et b(p+1:n).
    % T(p+1:n,p+1:n) est générée aléatoirement avec des coefficient
    % compris entre -1 et 1.
    T(p+1:n,p+1:n) = triu(ones(p-1)-2*rand(p-1));
```

```
% b(p+1:n) est le résultat du produit T(p+1:n,p+1:n)*ones(p-1,1)
% calculé avec une grande précision.
b(p+1:n) = mpfr_gemv(T(p+1:n,p+1:n), ones(p-1, 1), 256);
```

Sur la figure 9.2, nous reportons $\text{cond}(T,x)$ en fonction de $\text{cond}(T)$ pour les 1000 systèmes triangulaires ainsi générés. Comme cela a déjà été dit, on constate que la méthode (II) permet effectivement de générer des systèmes pour lesquels $\text{cond}(T,x)$ atteint 10^{35}. En particulier, $\text{cond}(T,x)$ reste voisin de $\text{cond}(T)$ (à quelques ordres de grandeur près), tant que $\text{cond}(T)$ est inférieur à 10^{30}. Néanmoins, pour des valeurs de $\text{cond}(T)$ supérieures, on observe $\text{cond}(T,x)$ est souvent nettement inférieur à $\text{cond}(T)$.

9.6.2 Précision effective

Rappelons que l'on travaille ici en double précision IEEE-754. Sur la figure 9.3, nous comparons la précision de la solution calculée par l'algorithme de substitution compensée CompTRSV (algorithme 9.9) à celle obtenue avec l'algorithme de substitution classique exécuté en double précision IEEE-754, c'est à dire l'algorithme TRSV (algorithme 9.1).

Nous utilisons également dans nos comparaisons la fonction BLAS_dtrsv_x de la bibliothèque XBLAS[2] [57], : cette fonction utilise l'arithmétique double-double, permettant ainsi la résolution de systèmes triangulaires par substitution, avec une précision de travail doublée et arrondi final de la solution calculée en double précision IEEE-754. La précision de la solution $\widehat{x}_d$ calculée par BLAS_dtrsv_x satisfait ainsi la borne d'erreur (9.9), que nous rappelons ici :

$$\frac{\|\widehat{x}_d - x\|_\infty}{\|\widehat{x}\|_\infty} \leq \mathbf{u} + n\mathbf{u}^2\,\text{cond}(T,x) + \mathcal{O}(\mathbf{u}^3). \tag{9.28}$$

On s'intéresse à la précision $\|\widehat{x} - x\|_\infty / \|x\|_\infty$ de la solution approchée $\widehat{x}$ calculée par chacun de ces algorithmes pour le système triangulaire $Tx = b$. Pour les 1000 systèmes triangulaire générés par la méthode (I), on reporte sur la figure 9.3 la précision de chaque solution calculée, en fonction du conditionnement de Skeel $\text{cond}(T,x)$. On procède de même sur la figure 9.4 pour 1000 systèmes triangulaires générés par la méthode (II).

Nous représentons également sur les figures 9.3 et 9.4 les bornes d'erreurs *a priori* (9.8) et (9.28) pour la précision du résultat calculé respectivement par TRSV et BLAS_dtrsv_x. Pour l'instant, nous ne nous intéressons pas à la borne d'erreur (9.22) pour CompTRSV, qui sera interprétée dans la section suivante.

Dans ces expériences numériques, on constate bien sûr que l'erreur relative entachant la solution calculée par l'algorithme de substitution TRSV est toujours supérieure à l'unité d'arrondi. En particulier, pour un conditionnement $\text{cond}(T,x)$ supérieur à $\mathbf{u}^{-1}$, cette erreur relative devient plus grande que 1 (ce que l'on représente sur nos graphique en la fixant à la valeur 1).

D'autre part, on observe que CompTRSV fournit en pratique un résultat aussi précis que BLAS_dtrsv_x. En particulier, tant que $\text{cond}(T,x)$ est inférieur à $\mathbf{u}^{-1}$, ces deux algorithmes calculent chacun une solution approchée dont la précision est de l'ordre de l'unité d'arrondi $\mathbf{u}$. On constate également que ces deux algorithmes présentent le même comportement numérique lorsque $\text{cond}(T,x)$ devient plus grand que $\mathbf{u}^{-1}$. En outre, on peut remarquer que,

[2]Bibliothèque disponible à l'adresse `http://crd.lbl.gov/~xiaoye/`, et dans une version plus récente à l'adresse `http://www.cs.berkeley.edu/~yozo/`

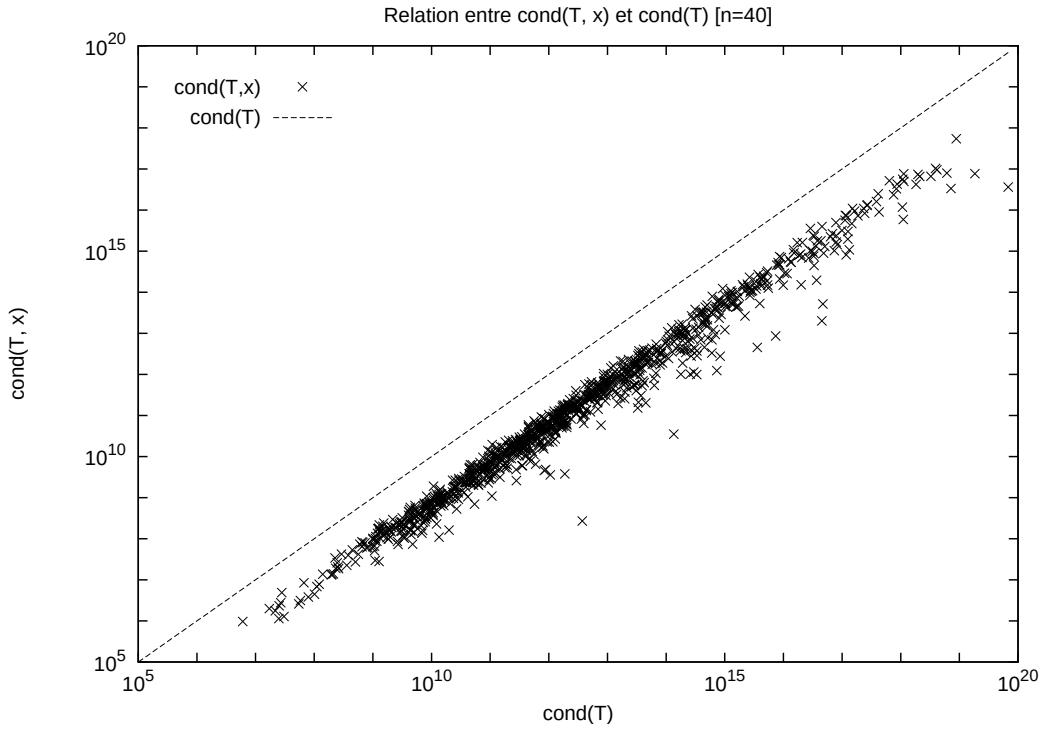


FIG. 9.1 – Relation entre $\mathrm{cond}(T, x)$ et $\mathrm{cond}(T)$ — méthode (I).

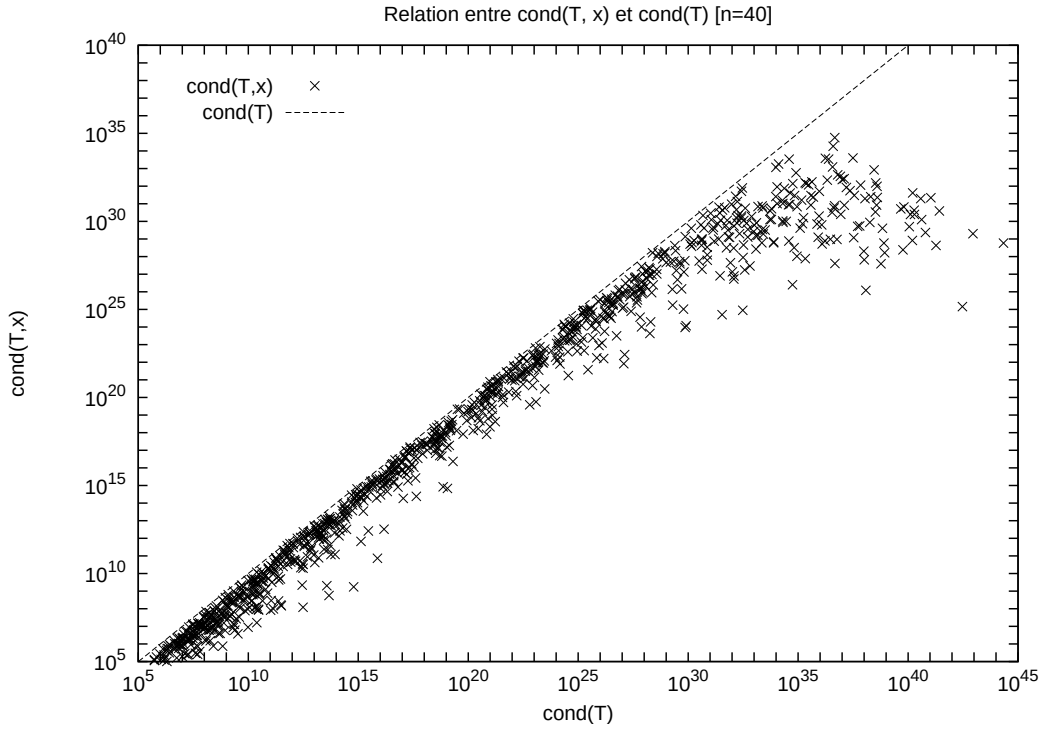


FIG. 9.2 – Relation entre $\mathrm{cond}(T, x)$ et $\mathrm{cond}(T)$ — méthode (II).

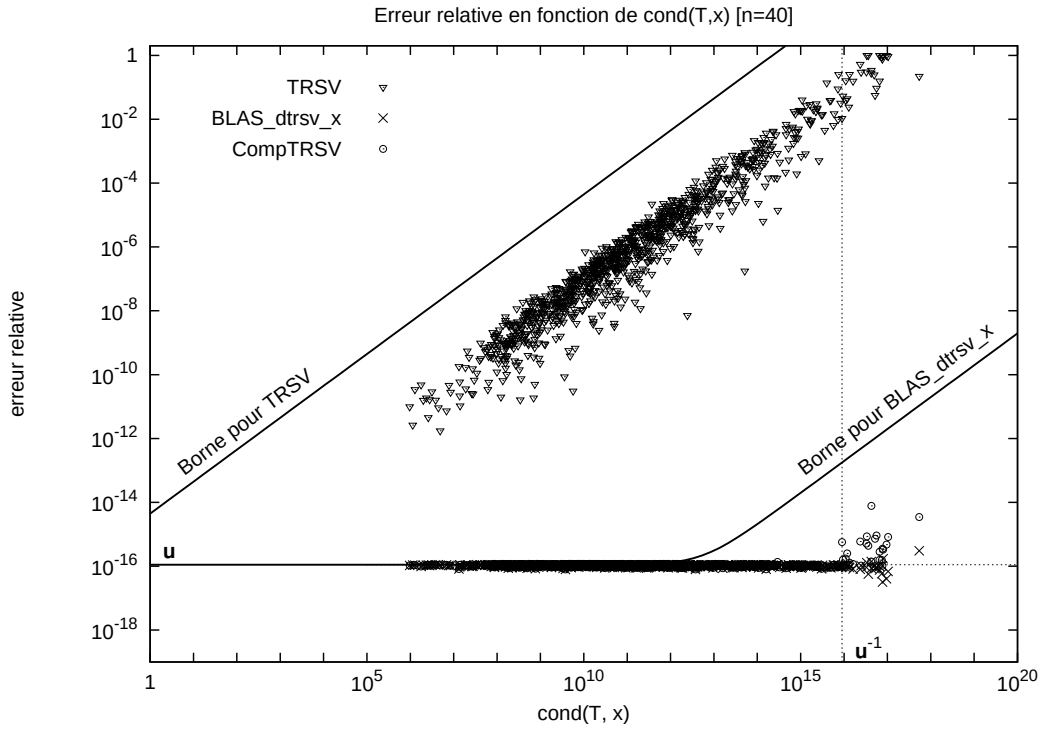


FIG. 9.3 – Précision effective de TRSV, CompTRSV et BLAS_dtrsv_x — méthode (I).

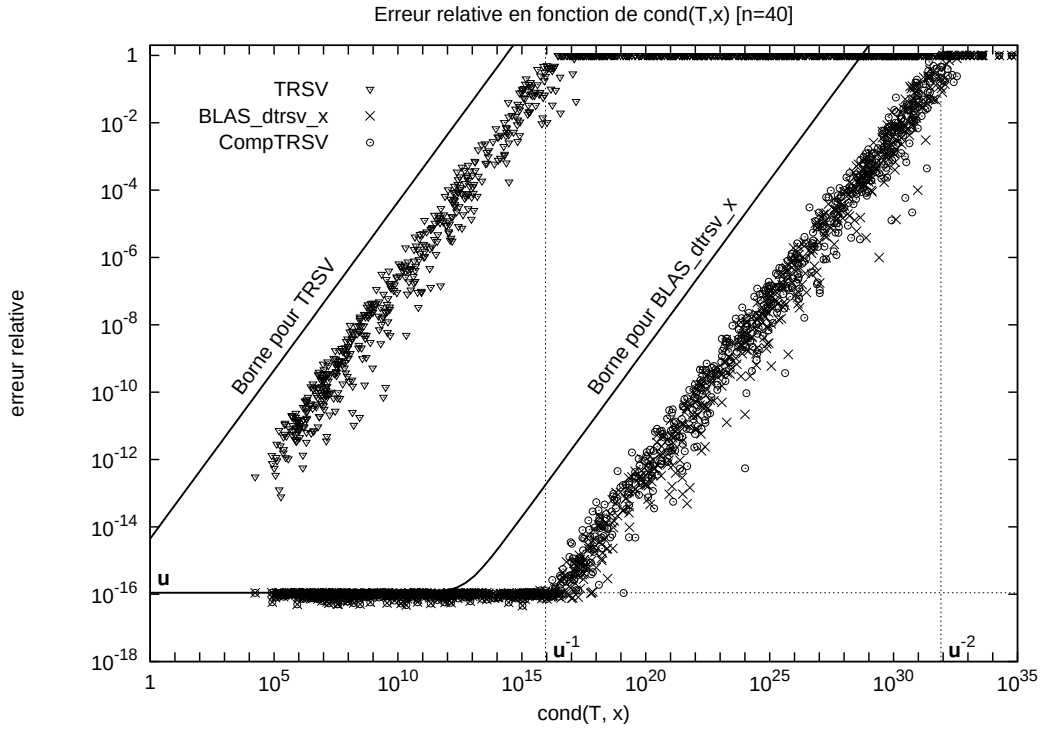


FIG. 9.4 – Précision effective de TRSV, CompTRSV et BLAS_dtrsv_x — méthode (II).

dans ces expériences, la borne d'erreur (9.9) satisfaite par BLAS_dtrsv_x l'est également par CompTRSV.

9.6.3 Interprétation de la borne d'erreur *a priori* pour CompTRSV

Rappelons dans un premier temps la borne d'erreur *a priori* (9.22) qui a été obtenue pour la précision de la solution compensée calculée par l'algorithme CompTRSV (algorithme 9.9) :

$$\frac{\|\overline{x} - x\|_\infty}{\|x\|_\infty} \leq \mathbf{u} + 2n(3n+1)\mathbf{u}^2\,\mathrm{K}(T,x) + \mathcal{O}(\mathbf{u}^3). \tag{9.29}$$

Cette borne d'erreur fait intervenir le facteur $\mathrm{K}(T,x)$ tel que

$$\mathrm{K}(T,x) = \frac{\|(|T^{-1}||T|)^2|x|\|_\infty}{\|x\|_\infty}.$$

Comme cela a déjà été dit, puisque la borne d'erreur (9.29) fait intervenir ce facteur $\mathrm{K}(T,x)$, elle ne permet pas de conclure que la solution compensé $\overline{x}$ est aussi précise que si elle avait été calculée par l'algorithme de substitution en précision doublée. D'autre part, nous avons vu que la meilleure majoration de $\mathrm{K}(T,x)$ en fonction de $\mathrm{cond}(T,x)$ que l'on peut écrire est la suivante :

$$\mathrm{K}(T,x) \leq \mathrm{cond}(T)\,\mathrm{cond}(T,x).$$

Néanmoins, nous montrons ci-dessous que, dans nos expériences, cette majoration est extrêmement pessimiste, et que $\mathrm{K}(T,x)$ peut être majoré en pratique par $\alpha\,\mathrm{cond}(T,x)$, avec ici $\alpha = 10^5$.

Pour ce faire, nous reportons $\mathrm{K}(T,x)$ en fonction de $\mathrm{cond}(T,x)$ sur le graphique de la figure 9.5, pour les 1000 systèmes triangulaires générés par la méthode (I). Nous représentons également $\mathrm{cond}(T)\,\mathrm{cond}(T,x)$ et $10^5\,\mathrm{cond}(T,x)$ en fonction de $\mathrm{cond}(T,x)$. On effectue les mêmes tracés sur la figure 9.6 pour les systèmes générés par la méthode (II). Au travers de ces résultats, on observe que $\mathrm{cond}(T)\,\mathrm{cond}(T,x)$ constitue comme annoncé une majoration extrêmement pessimiste de $\mathrm{K}(T,x)$: reporter cette majoration de $\mathrm{K}(T,x)$ dans la borne d'erreur (9.29) lui ôterait tout intérêt pratique. D'autre part, il est aisé de constater que, pour l'ensemble des systèmes triangulaires que nous avons générés, on a

$$\mathrm{K}(T,x) \leq \alpha\,\mathrm{cond}(T,x), \quad \text{avec} \quad \alpha = 10^5.$$

Nous n'affirmons pas que cette majoration de $\mathrm{K}(T,x)$ soit valable pour tout système triangulaire $Tx = b$, ni qu'elle est indépendante de la dimension du système considéré. Nous l'utilisons ici simplement pour interpréter la borne d'erreur (9.29) dans le cadre de nos expériences numériques. En effet, en reportant $\mathrm{K}(T,x) \leq \alpha\,\mathrm{cond}(T,x)$ dans la borne (9.29), on obtient

$$\frac{\|\overline{x} - x\|_\infty}{\|x\|_\infty} \leq \mathbf{u} + \alpha 2n(3n+1)\mathbf{u}^2\,\mathrm{cond}(T,x) + \mathcal{O}(\mathbf{u}^3). \tag{9.30}$$

Sur la figure 9.7, nous reportons en fonction de $\mathrm{cond}(T,x)$ la précision de la solution calculée par CompTRSV pour les 1000 systèmes générés par la méthode (I). Les mêmes tracés sont effectués sur la figure 9.8 pour les 1000 systèmes générés par la méthode (II). Dans les deux cas, nous représentons également :

- les valeurs de la borne (9.29), ce qui donne évidement un nuage de points, puisque $\mathrm{K}(T,x)$ intervient explicitement dans cette borne ;

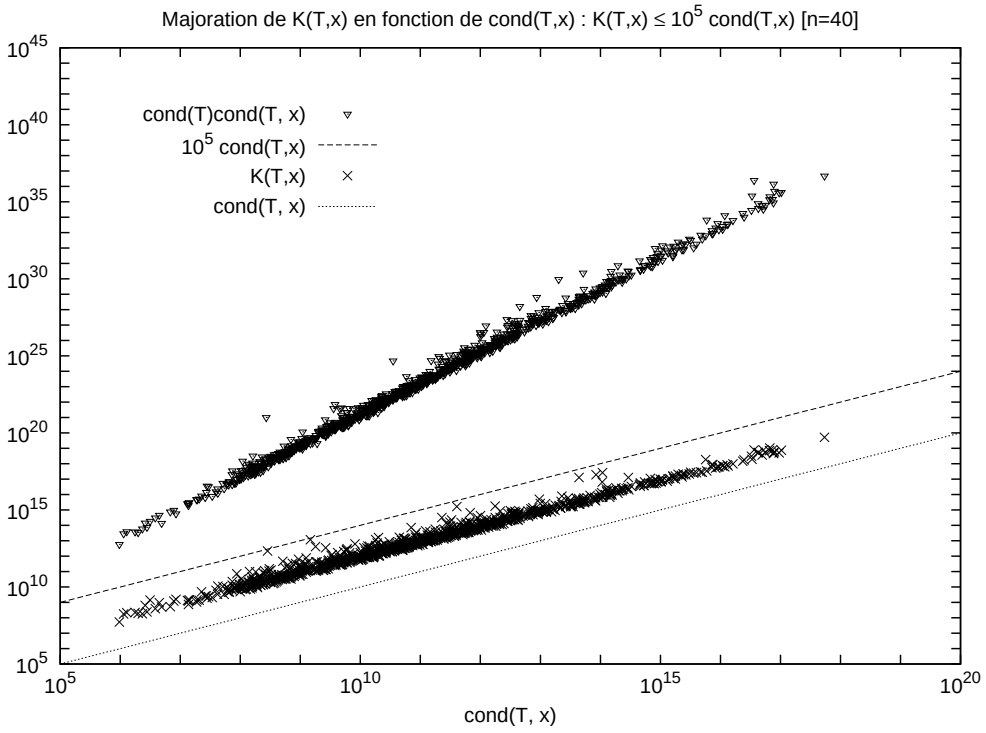


FIG. 9.5 – Majoration de $\mathrm{K}(T,x)$ en fonction de $\mathrm{cond}(T,x)$ — méthode (I).

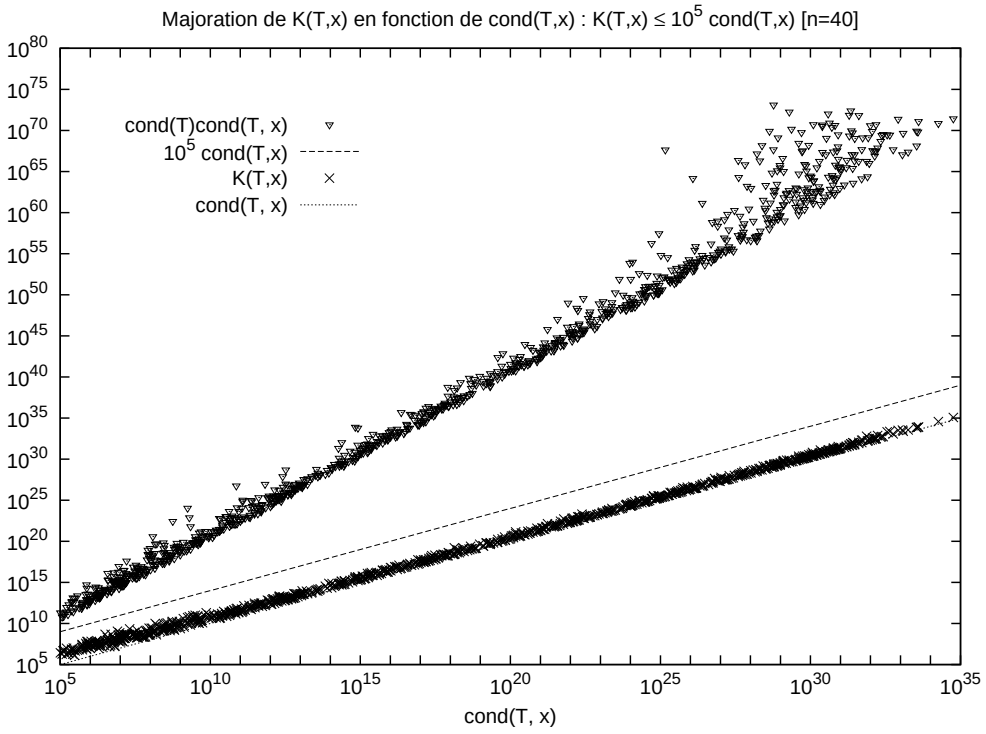


FIG. 9.6 – Majoration de $\mathrm{K}(T,x)$ en fonction de $\mathrm{cond}(T,x)$ — méthode (II).

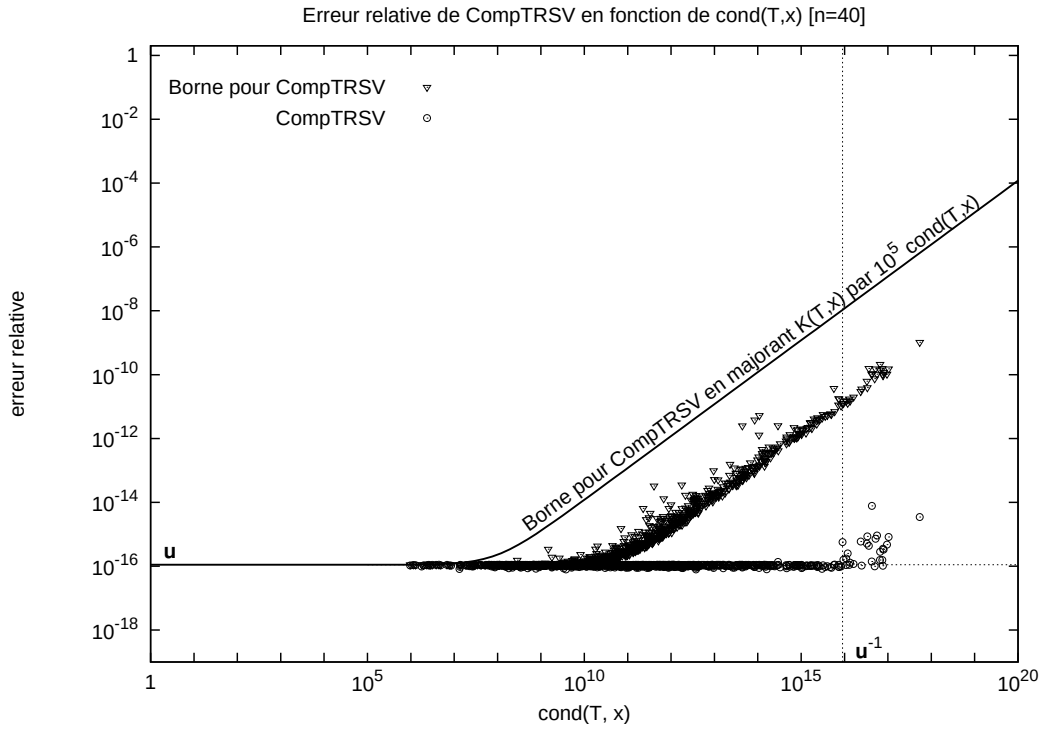


FIG. 9.7 – Interprétation de la borne d'erreur pour CompTRSV — méthode (I).

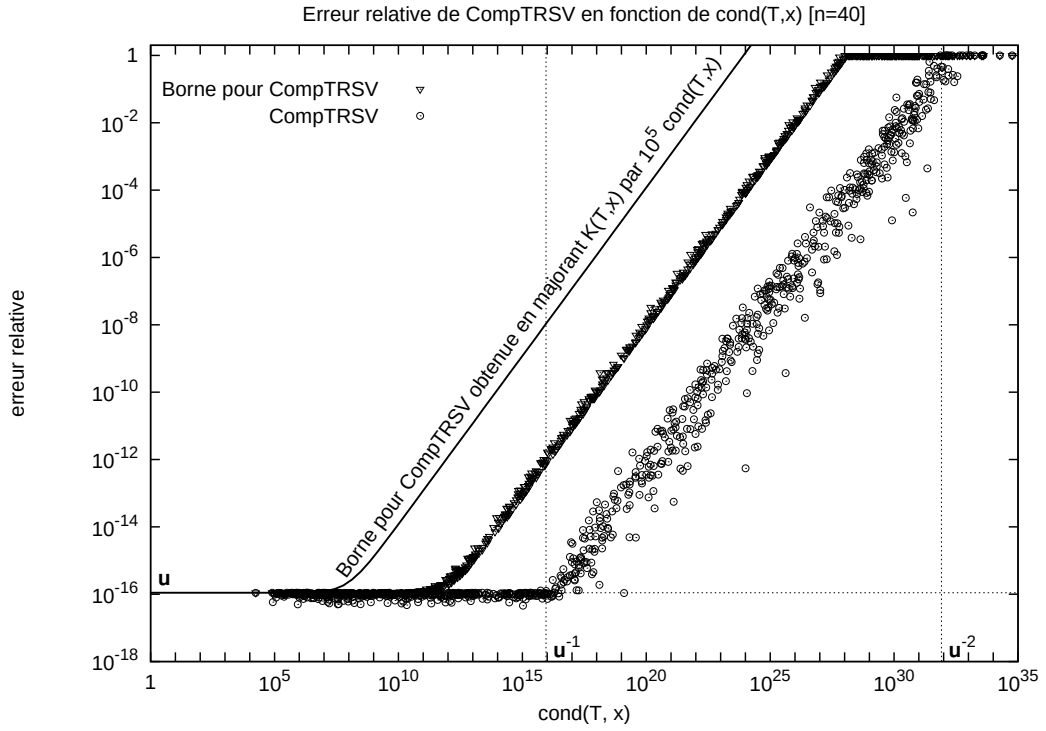


FIG. 9.8 – Interprétation de la borne d'erreur pour CompTRSV — méthode (II).

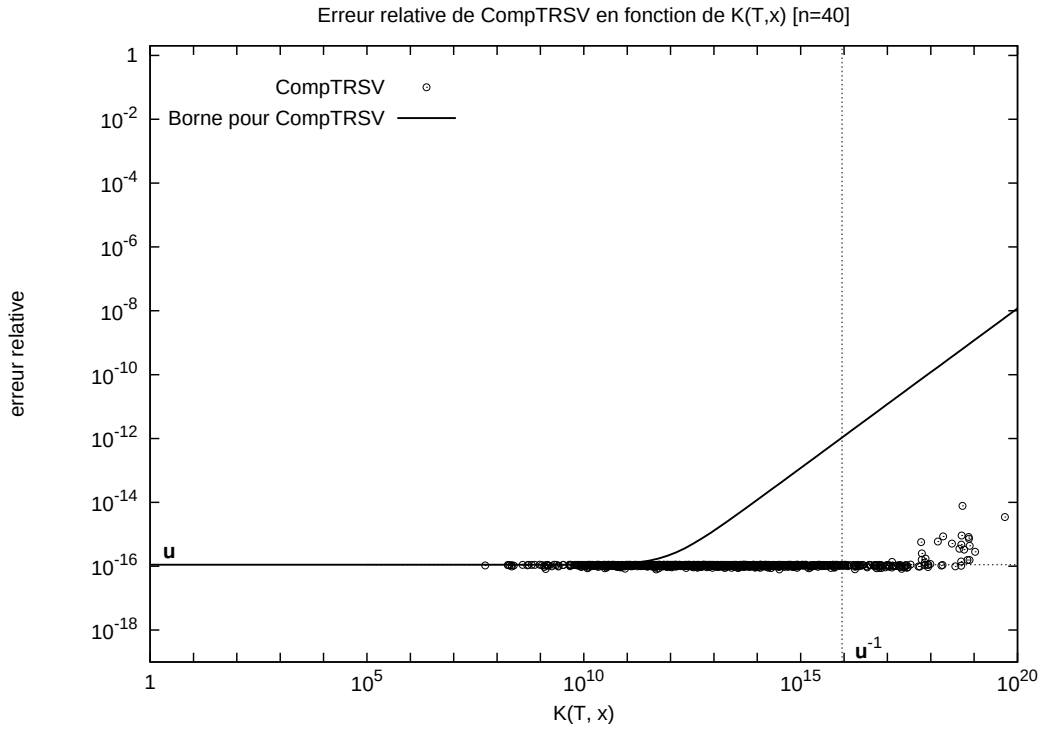


FIG. 9.9 – Interprétation de la borne d'erreur pour CompTRSV — méthode (I).

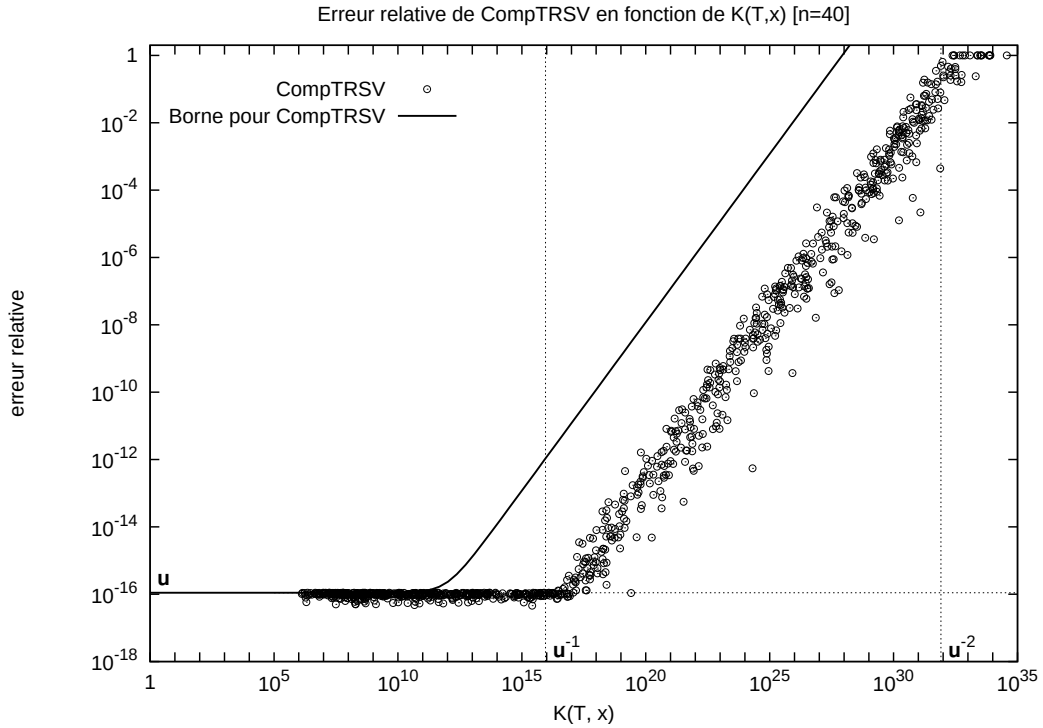


FIG. 9.10 – Interprétation de la borne d'erreur pour CompTRSV — méthode (II).

– la borne (9.30), qui apparaît sous la forme d'une courbe, puisque $\mathrm{K}(T,x)$ a été majoré par $\alpha\,\mathrm{cond}(T,x)$.

On constate certes dans les deux cas que les bornes d'erreurs (9.29) et (9.30) restent très pessimistes. Néanmoins, la borne d'erreur (9.30) indique, dans le cadre de ces expériences, que la précision de la solution compensée est de l'ordre de l'unité d'arrondi $\mathbf{u}$ tant que le conditionnement est inférieur à 10^7.

Finalement, nous reportons sur les figures 9.9 et 9.10 la précision de chaque solution compensée calculée, en fonction de $\mathrm{K}(T,x)$. Nous représentons sur ces figures la borne d'erreur (9.29) pour la solution calculée par CompTRSV. Naturellement on constate que la précision de la solution compensée est de l'ordre de l'unité d'arrondi tant que $\mathrm{K}(T,x)$ est inférieur à $\mathbf{u}^{-1}$ — est même pour des valeurs légèrement plus grandes, puisque $\mathrm{cond}(T,x) \leq \mathrm{K}(T,x)$. On constate également que la borne d'erreur (9.29) assure effectivement une erreur relative de l'ordre de l'unité d'arrondi tant que $\mathrm{K}(T,x)$ est inférieur à 10^{11}.

9.7 Tests des performances de CompTRSV

Dans cette section, nous nous intéressons au surcoût introduit en pratique par l'algorithme CompTRSV (algorithme 9.9), par rapport à l'algorithme de substitution classique TRSV (algorithme 9.1). Tous nos tests sont effectués en utilisant la double précision IEEE-754 comme précision de travail.

Afin de mettre en évidence les performances pratiques de CompTRSV, nous utiliserons comme référence la fonction BLAS_dtrsv_x de la bibliothèque XBLAS [57], déjà utilisé dans la section précédente.

Nous utilisons les mêmes techniques de programmation pour CompTRSV que celles utilisées dans la bibliothèque XBLAS : en particulier, notre algorithme est implanté simplement en langage C, sans blocage des données [57]. Il est également à noter que l'implantation de l'algorithme de substitution TRSV que nous utilisons pour nos tests est celle fournie avec la bibliothèque XBLAS (fonction BLAS_dtrsv).

Nos mesures de performances sont effectuées dans les environnements (III), (IV), (V) et (VI) déjà listés au chapitre 5 (tableau 5.1), et que nous rappelons dans le tableau 9.1 ci-dessous.

Tab. 9.1 – Environnements expérimentaux.

environnent	description
(II)	Intel Pentium 4, 3.0GHz, GNU Compiler Collection 4.1.2, fpu sse
(IV)	Intel Pentium 4, 3.0GHz, Intel C Compiler 9.1, fpu sse
(V)	AMD Athlon 64, 2 GHz, GNU Compiler Collection 4.1.2, fpu sse
(VI)	Itanium 2, 1.5 GHz, GNU Compiler Collection 4.1.1

Il est à noter que la bibliothèque XBLAS n'a pas été conçue pour tirer partie de l'opérateur FMA disponible sur l'architecture Itanium : c'est la transformation exacte TwoProd (algorithme 3.9) qui est utilisée sur cette architecture pour l'implantation de l'arithmétique double-double, au lieu de TwoProdFMA (algorithme 3.11) (voir chapitre 6). Afin d'assurer une comparaison juste de CompTRSV et BLAS_dtrsv_x dans l'environnement (VI), nous n'utilisons pas TwoProdFMA pour l'implantation de CompTRSV.

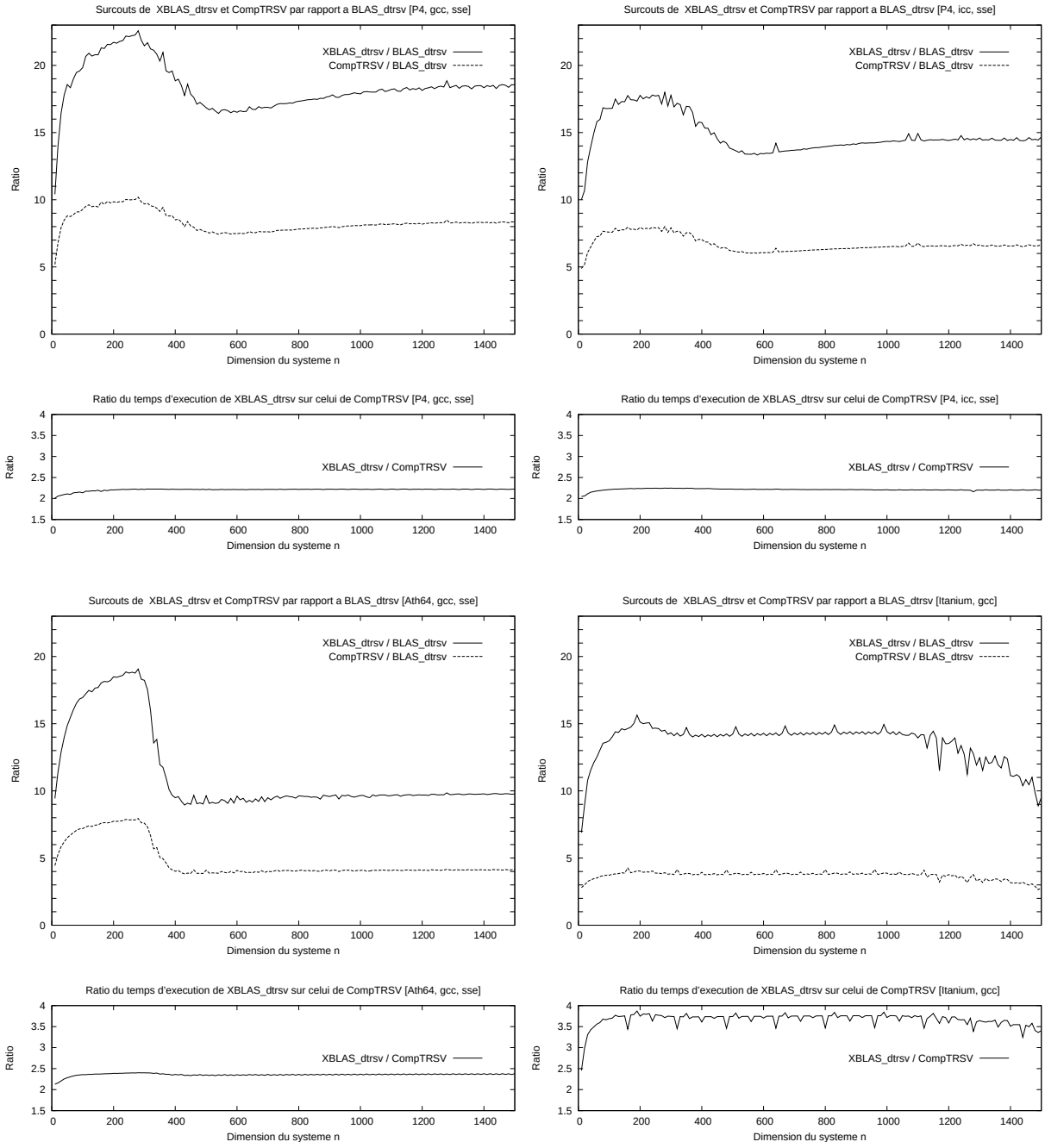


FIG. 9.11 – Résultats des tests de performance.

Sur la figure 9.11, nous représentons dans chaque environnement expérimental le ratio du temps d'exécution de CompTRSV sur celui de TRSV, la dimension n du système triangulaire à résoudre variant de 10 à 1500 par pas de 10 : ce ratio représente, en fonction de n, le surcoût introduit par CompTRSV par rapport à l'algorithme classique TRSV. Nous reportons de même le surcoût introduit par BLAS_dtrsv_x en fonction de n. Enfin, nous représentons également le ratio du temps d'exécution de BLAS_dtrsv_x sur celui de CompTRSV.

Que ce soit dans le cas de CompTRSV ou dans celui de BLAS_dtrsv_x, on constate que

le surcoût introduit par ces algorithmes est très variable en fonction de n, mais également en fonction de l'environnement expérimental considéré. Cela est dû aux échanges de données entre le processeur et la mémoire. En particulier, le surcoût introduit par le chargement dans les registres du processeur des éléments de la matrice varie en fonction de la taille de celle-ci. De plus, tous les environnements utilisés ici ne présentent pas la même hiérarchie mémoire (notamment, la taille des caches est variable).

Le décompte des opérations flottantes dans le code de BLAS_dtrsv_x montre qu'un appel à cette fonction effectue $45n^2/2 + \mathcal{O}(n)$ opérations flottantes. Comme on l'a déjà vu, ce même décompte donne $27n^2/2 + \mathcal{O}(n)$ opérations flottantes pour CompTRSV. Comme dans le cas du schéma de Horner compensé (voir chapitre 5), on observe que les surcoûts mesurés sont toujours plus faibles que ceux auxquels on aurait pu s'attendre d'après le décomptes des opérations flottantes. En effet, si l'on ne considère que ces décomptes,

$$\frac{\text{CompTRSV}}{\text{TRSV}} \approx \frac{27}{2} = 13.5, \quad \text{et} \quad \frac{\text{BLAS\_dtrsv\_x}}{\text{TRSV}} \approx \frac{45}{2} = 22.5.$$

On notera néanmoins que BLAS_dtrsv_x atteint effectivement en pratique un surcoût de l'ordre de 22.5 dans l'environnement (III).

Remarquons finalement que le ratio du temps d'exécution de BLAS_dtrsv_x sur celui de CompTRSV reste relativement stable dans les différents environnement étudiés. On retiendra donc que CompTRSV s'exécute toujours significativement plus rapidement que BLAS_dtrsv_x : environ 2 fois plus rapidement dans les environnements (III), (IV) et (V) ; de l'ordre de 3.5 fois plus rapidement dans l'environnement (V).

9.8 Conclusion

Dans ce chapitre, nous nous sommes intéressés à la résolution compensée de systèmes triangulaires, en montrant comment compenser l'effet des erreurs d'arrondi générées lors du calcul d'une solution par substitution. Nous avons vu que le fait de calculer un terme correctif $\widehat{c}$, puis une solution compensée $\overline{x} = \widehat{x} \oplus \widehat{c}$ s'avère équivalent à l'utilisation d'une étape de raffinement itératif pour améliorer la précision de la solution initiale $\widehat{x}$, calculée par substitution. Nous avons donc envisagé deux méthodes de calcul du résidu, chacune donnant lieu à un algorithme compensé pour la résolution de systèmes triangulaires.

Dans l'algorithme CompTRSV, le résidu est calculé à l'aide des transformation exactes pour les opérations élémentaires. Nous avons en effet montré que le résidu peut être exprimé exactement en fonction des erreurs d'arrondis générées par l'algorithme de substitution. Dans CompTRSV2, le résidu est calculé de façon classique à l'aide d'un algorithme de produit scalaire en précision doublée, à savoir l'algorithme Dot2 de Ogita, Rump et Oishi [70]. Nous avons montré que les deux méthodes de calcul du résidu envisagées sont en fait quasi-équivalentes : ces deux méthodes de calcul d'un résidu approché différent seulement par l'ordre de quelques opérations arithmétiques, et présentent des bornes d'erreurs similaires. Par conséquent, les algorithmes CompTRSV et CompTRSV2 sont très similaires, et nous nous sommes intéressé à CompTRSV dans nos expériences numériques.

Bien que nous ne soyons pas parvenu à démontrer, à l'aide d'une borne d'erreur *a priori*, le fait que la solution calculée par CompTRSV soit aussi précise que si elle avait été calculée en précision doublée, on peut clairement observer ce comportement numérique en pratique. En particulier, la précision relative de la solution compensée est en pratique de

l'ordre de l'unité d'arrondi tant que le conditionnement de Skeel du système triangulaire est inférieur à $\mathbf{u}^{-1}$. De plus, même si notre analyse d'erreur de CompTRSV ne permet pas de conclure à un doublement de la précision des calculs intermédiaires, nous avons montré expérimentalement comment interpréter la borne obtenues sur l'erreur entachant le résultat compensé.

En outres, nos tests de performances montrent que CompTRSV est en pratique au moins deux fois plus rapide que la fonction de résolution de systèmes triangulaires de la bibliothèque XBLAS, tout en simulant également un doublement de la précision des calculs. Cela justifie à nouveau l'intérêt pratique de l'algorithme compensé CompTRSV face à son équivalent en arithmétique double-double.

Chapitre 10

Conclusion et perspectives

10.1 Contributions

Dans cette thèse, nous nous sommes intéressés à la problématique suivante : comment améliorer et valider la précision d'un résultat calculé en arithmétique flottante, tout en conservant de bonnes performances pratiques ? En utilisant la compensation des erreurs d'arrondi comme méthode privilégiée pour améliorer la précision des calculs, nous avons apporté des éléments de réponse à cette problématique. Nous résumons ci-dessous nos principaux résultats.

Amélioration de la précision du résultat.

La précision du résultat d'un calcul en précision finie dépend de trois facteurs : le conditionnement du problème à résoudre, la stabilité de l'algorithme numérique et la précision de l'arithmétique utilisée. Cette dernière est souvent l'arithmétique flottante binaire IEEE-754 qui permet une précision maximale de l'ordre de 16 chiffres décimaux en double précision. Cette précision n'est pas suffisante si l'on cherche une solution précise de problèmes mal conditionnés. Un certain nombre de solutions classiques permettent d'augmenter la précision, comme par exemple l'utilisation de l'arithmétique double-double.

Nous nous sommes plus particulièrement intéressés à l'évaluation polynomiale par la méthode de Horner. Notre objectif a été de proposer une alternative aux double-doubles qui soit à la fois performante, fiable et portable. Nous avons proposé une version compensée du schéma de Horner. L'évaluation effectuée par cet algorithme est aussi précise que celle calculée par le schéma de Horner classique avec une précision double de la précision courante. De plus, le schéma de Horner compensé se montre en pratique deux fois plus rapide que son homologue basé sur les double-doubles. Nous avons donné en outre une condition suffisante, portant sur le nombre de conditionnement de l'évaluation, permettant d'assurer que le résultat produit par le schéma de Horner compensé est un arrondi fidèle du résultat exact.

Nous avons ensuite proposé une version du schéma de Horner dans laquelle les erreurs d'arrondis sont compensées $K-1$ fois. L'algorithme compensé correspondant produit un résultat aussi précis que s'il avait été calculé par le schéma de Horner classique en K fois la la précision de travail ($K \geq 2$).

La résolution de systèmes linéaires triangulaires nous a également servi d'exemple quant à l'utilisation de la compensation des erreurs d'arrondi pour améliorer la précision. Nous avons montré comment compenser les erreurs d'arrondi générées par l'algorithme de substitution, et mis en évidence le lien qui existe entre cette méthode et l'utilisation d'une étape de raffinement itératif en précision doublée. Nos expériences numériques indiquent que la solution compensée est à nouveau aussi précise que si elle avait été calculée par substitution en précision doublée, puis arrondie vers la précision de travail. Notre analyse d'erreur ne permet pas de garantir ce comportement numérique, mais indique que la précision de la solution compensée est effectivement améliorée.

Valider la qualité du résultat

Nous avons montré comment valider la précision du résultat compensé au travers de l'exemple du schéma de Horner compensé. Le calcul de la borne d'erreur *a posteriori* que nous avons proposé pour le résultat de l'évaluation compensée, ne repose que sur des opérations élémentaires en arithmétique flottante, dans le mode d'arrondi au plus proche : ni bibliothèque de calcul d'intervalles, ni changement du mode d'arrondi ne sont utilisés. Nous proposons également un test permettant de déterminer dynamiquement si l'évaluation compensée produit un arrondi fidèle de l'évaluation exacte.

La borne d'erreur *a posteriori* est effectivement bien plus fine que celle obtenue via l'analyse d'erreur *a priori* du schéma de Horner. Précisons que cette borne d'erreur *a posteriori* prend soigneusement en compte les erreurs d'arrondi commises lors de son évaluation : nous avons prouvé rigoureusement qu'il s'agit bien d'une borne sur l'erreur directe, et non pas d'une simple estimation.

En utilisant les mêmes techniques, nous avons également procédé à la validation du schéma de Horner compensé $K-1$ fois.

Dans les deux cas, nos tests de performances montrent que le coût de la validation de l'évaluation compensée est en pratique très faible : le temps d'exécution moyen observé n'est qu'au plus 1.5 fois plus élevé que celui de l'algorithme non validé.

Performances des algorithmes compensés.

L'intérêt pratique des algorithmes compensés face aux autres solutions logicielles permettant de simuler une précision équivalente a été pleinement justifié au travers de nos tests de performances. Dans le cas d'un doublement de la précision, les algorithmes compensés décrits dans cette thèse s'exécutent toujours au moins deux fois plus rapidement que leurs homologues basés sur l'arithmétique double-double, solution pourtant considérée comme une référence en terme de performances [57].

Nous avons clairement montré que le schéma de Horner compensé présente plus de parallélisme d'instructions que le schéma de Horner en arithmétique double-double. Nous expliquons ainsi de façon qualitative les bonnes performances pratiques du schéma de Horner compensé sur les architectures superscalaires modernes, qui sont précisément conçues pour tirer partie de ce parallélisme d'instructions.

Chaque opération arithmétique faisant intervenir un double-double se termine par une étape de renormalisation [81, 57]. Nous avons également montré que le défaut de parallé-

lisme d'instructions dans le schéma de Horner avec les double-doubles est précisément dû à ces étapes de renormalisation.

Les conclusions de cette étude se généralisent facilement à tous les autres algorithmes compensés décrits dans cette thèse. De plus, les bonnes performances pratiques de la sommation et du produit scalaire compensés étaient effectivement observées dans [70], mais restaient inexpliquées par les auteurs de cet article. Nous avons montré que l'absence d'étapes de renormalisation dans ces algorithmes compensés explique de façon qualitative leur efficacité pratique face à leurs équivalents en arithmétique double-double.

10.2 Perspectives

Nous présentons ci-dessous quelques perspectives quant à ce travail de thèse.

Amélioration de la précision et validation

La compensation des erreurs d'arrondi permet d'obtenir des algorithmes plus précis, qui sont également plus efficaces que ceux basés sur les solution génériques de l'état de l'art pour améliorer la précision de calcul. En particulier, la compensation s'avère être une alternative intéressante en pratique face à l'utilisation de l'arithmétique double-double. Nous avons illustré cela au travers de deux exemples : l'évaluation de polynômes par le schéma de Horner compensé et la résolution compensée de système triangulaires. Il nous semble possible dans un premier temps de s'intéresser à la généralisation de ces deux exemples.

Le généralisation du schéma de Horner compensé peut se faire en proposant un algorithme permettant d'évaluer plus précisément, non seulement un polynôme, mais aussi ses dérivées. Une application de cet algorithme serait par exemple la recherche des racines de polynômes à coefficients flottants. Le schéma de Horner compensé peut également être généralisé pour l'évaluation de polynômes à plusieurs variables [77].

Il convient également de poursuivre l'étude de la résolution compensée de systèmes linéaires triangulaires. En effet, l'analyse d'erreur directe effectuée dans cette thèse ne permet pas d'expliquer clairement le comportement numérique de la solution compensée, tel que nous l'observons expérimentalement. Une meilleure compréhension des effets de la compensation passe par une étude théorique plus approfondie, mais aussi par des expériences numériques plus poussées. En particulier, il serait intéressant de pratiquer des tests intensifs, comme ceux mis en place par les auteurs de [25] pour mettre en évidence le comportement numérique de l'algorithme de raffinement itératif qu'ils proposent. De plus, de nombreux articles [73, 72, 75, 71, 76] montrent comment valider la solution d'un système linéaire calculée en précision finie. Il serait donc également intéressant de valider la résolution compensée de systèmes triangulaires.

Nous avons vu que calculer une solution compensée pour un système triangulaire revient à utiliser une étape de raffinement itératif pour améliorer la précision de la solution calculée par substitution. En procédant de la même manière, il est clairement possible de s'intéresser à la résolution compensée de systèmes linéaires quelconques. En supposant que la solution initiale est obtenue par élimination Gaussienne, et calcul du résidu effectué en

précision doublée, que dire de la qualité de la solution compensée ? À nouveau, la qualité de la solution compensée pourra être étudiée au travers d'une étude théorique puis expérimentale, et également garantie via des méthodes relevant de la validation.

Performance des algorithmes compensés

De nouvelles plates-formes dédiées au calcul scientifique, et notamment au calcul flottant, apparaissent aujourd'hui. Citons tout d'abord le domaine du *General Purpose Processing on Graphics Processing Units* (GPGPU). L'idée du GPGPU est d'exploiter pour le calcul scientifique la puissance des processeurs graphiques actuels. Les dernières génération de processeurs graphiques disposent en effet de nombreuses unités de calcul flottant capables de travailler en parallèle, sur un format similaire à la simple précision IEEE-754. Le processeur Cell d'IBM constitue également un exemple d'architecture émergente qui présente un énorme potentiel pour le calcul scientifique, avec une performance crête de plus de 200 Gflop/s pour le calcul en simple précision IEEE-754.

Comme nous l'avons vu, les performances des algorithmes compensés se montrent excellentes sur les architectures superscalaires largement disponibles actuellement. Qu'en est-il sur les architectures émergentes citées ci-dessus ? Pour apporter une réponse à cette question, il convient de travailler sur l'implantation efficace des algorithmes compensés dans ces nouveaux environnements.

Bibliographie

[1] Gmp : GNU Multiple Precision arithmetic library. Available at `http://gmplib.org/`.

[2] I. J. Anderson. A distillation algorithm for floating-point summation. *SIAM J. Sci. Comput.*, 20(5) :1797–1806, 1999.

[3] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4) :345–420, 1994.

[4] David H. Bailey. A Fortran-90 double-double library. Available at `http://crd.lbl.gov/~dhbailey/mpdist/`.

[5] David H. Bailey. A Fortran 90-based multiprecision system. *ACM Trans. Math. Softw.*, 21(4) :379–387, 1995.

[6] David H. Bailey. High-precision floating-point arithmetic in scientific computation. *Computing in Science and Engineering*, 7(3) :54–61, May/June 2005.

[7] David H. Bailey, Yozo Hida, Karthik Jeyabalan, Xiaoye S. Li, and Brandon Thompson. C++/Fortran-90 arbitrary precision package. Available at `http://crd.lbl.gov/~dhbailey/mpdist/`.

[8] Å. Björck. Iterative refinement and reliable computing. In M. G. Cox and S. J. Hammarling, editors, *Reliable Numerical Computation*, pages 249–266. Clarendon Press, Oxford, 1990.

[9] G. Bohlender, W. Walter, P. Kornerup, and D. W. Matula. Semantics for exact floating point operations. In Peter Kornerup and David W. Matula, editors, *Proceedings : 10th IEEE Symposium on Computer Arithmetic : June 26–28, 1991, Grenoble, France*, pages 22–26, pub-IEEE :adr, 1991. IEEE Computer Society Press.

[10] Sylvie Boldo. Pitfalls of a full floating-point proof : example on the formal proof of the veltkamp/dekker algorithms. In Ulrich Furbach and Natarajan Shankar, editors, *Third International Joint Conference on Automated Reasoning*, Lecture Notes in Computer Science, Seattle, USA, 2006. Springer-Verlag.

[11] Sylvie Boldo and Marc Daumas. Representable correcting terms for possibly underflowing floating point operations. In *ARITH '03 : Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH-16'03)*, page 79, Washington, DC, USA, 2003. IEEE Computer Society.

[12] Sylvie Boldo and Jean-Michel Muller. Some functions computable with a fused-mac. In IEEE, editor, *Proceedings of the 17th IEEE Symposium on Computer Arithmetic, 2005, Cape Cod, Massachusetts, USA*, Washington, DC, USA, 2005. IEEE Computer Society.

[13] Richard P. Brent. A fortran multiple-precision arithmetic package. *ACM Trans. Math. Softw.*, 4(1) :57–70, 1978.

[14] Keith Briggs. Doubledouble floating point arithmetic. Available at `http://keithbriggs.info/`, but no longer supported.

[15] Françoise Chaitin-Chatelin and Valérie Frayssé. *Lectures on finite precision computations*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1996.

[16] Marius Cornea, John Harrison, and Ping Tak Peter Tang. *Scientific computing on Itanium based systems*. Intel Press, 2002.

[17] A. Cuyt, P. Kuterna, B. Verdonk, and D. Verschaeren. Underflow revisited. *Calcolo*, 39 :169–179, 2002.

[18] Florent de Dinechin and G. Villard. High precision numerical accuracy in physics research. *Nuclear Inst. and Methods in Physics Research, A*, 559 :207–210, 2006.

[19] T. J. Dekker. A floating-point technique for extending the available precision. *Numer. Math.*, 18 :224–242, 1971.

[20] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, Philadelphia, PA, 1997.

[21] J. W. Demmel and Xiaoye Li. Faster numerical algorithms via exception handling. *IEEE Trans. Comput.*, 43(8) :983–992, 1994.

[22] James Demmel. Underflow and the reliability of numerical software. *SIAM Journal on Scientific and Statistical Computing*, 5(4) :887–919, 1984.

[23] James Demmel and Yozo Hida. Accurate and efficient floating point summation. *SIAM Journal on Scientific Computing*, 25(4) :1214–1248, 2003.

[24] James Demmel and Yozo Hida. Fast and accurate floating point summation with application to computational geometry. *Numerical Algorithms*, 37(1–4) :101–112, December 2004.

[25] James Demmel, Yozo Hida, William Kahan, Xiaoye S. Li, Sonil Mukherjee, and E. Jason Riedy. Error bounds from extra-precise iterative refinement. *ACM Trans. Math. Softw.*, 32(2) :325–351, 2006.

[26] T. O. Espelid. On floating-point summation. *SIAM Rev.*, 37 :603–607, 1995.

[27] G. H. Golub and C. van Loan. *Matrix Computations*. The Johns Hopkins University Press, London, 3 edition, 1996.

[28] Stef Graillat, Philippe Langlois, and Nicolas Louvet. Improving the compensated Horner scheme with a fused multiply and add. In *Proceedings of the 21st Annual ACM Symposium on Applied Computing*, volume 2, pages 1323–1327. Association for Computing Machinery, April 2006.

[29] Andreas Griewank. *Evaluating derivatives : principles and techniques of algorithmic differentiation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[30] IEEE 754 Revision Working Group. *Draft Standard for Floating-Point Arithmetic P754*. October 1985. Available at `http://754r.ucbtest.org/`.

[31] Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. The MPFR library. Available at `http://www.mpfr.org/`.

[32] John R. Hauser. Handling floating-point exceptions in numeric programs. *ACM Trans. Program. Lang. Syst.*, 18(2) :139–174, 1996.

[33] John L. Hennessy and David A. Patterson. *Computer Architecture – A Quantitative Approach.* Morgan Kaufmann Publishers, third edition, 2003.

[34] Yozo Hida, Xiaoye S. Li, and David H. Bailey. Double-double and quad double package. Available at `http://crd.lbl.gov/~dhbailey/mpdist/`.

[35] Yozo Hida, Xiaoye S. Li, and David H. Bailey. Algorithms for quad-double precision floating point arithmetic. In Neil Burgess and Luigi Ciminiera, editors, *Proceedings of the 15th Symposium on Computer Arithmetic*, pages 155–162, Vail, Colorado, 2001.

[36] Nicholas J. Higham. The accuracy of solutions to triangular systems. *SIAM Journal on Numerical Analysis*, 26(5) :1252–1265, 1989.

[37] Nicholas J. Higham. The accuracy of floating point summation. *SIAM J. Sci. Comput.*, 14(4) :783–799, July 1993.

[38] Nicholas J. Higham. Iterative refinement for linear systems and LAPACK. *IMA J. Numer. Anal.*, 17(4) :495–509, 1997.

[39] Nicholas J. Higham. *Accuracy and stability of numerical algorithms.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, second edition, 2002.

[40] IBM. *PowerPC User Instruction Set Architecture*, 2005. Book I.

[41] *IEEE Standard for binary floating-point arithmetic, ANSI/IEEE Standard 754-1985.* 1985.

[42] Intel. *Intel 64 an IA-32 Architectures Software Developer's Manual*, 2007.

[43] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2007. Volume 1 : Basic Architecture.

[44] Luc Jaulin, Michel Kieffer, Olivier Didrit, and Eric Walter. *Applied interval analysis.* Springer, 2001.

[45] W. Kahan. Pracniques : further remarks on reducing truncation errors. *Commun. ACM*, 8(1), 1965.

[46] Alan H. Karp and Peter Markstein. High-precision division and square root. *ACM Trans. Math. Softw.*, 23(4) :561–589, 1997.

[47] Donald E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms.* Addison-Wesley, Reading, MA, USA, third edition, 1998.

[48] U. W. Kulisch and W. L. Miranker. The arithmetic of the digital computer. *SIAM Rev.*, 28 :1–40, 1986.

[49] Philippe Langlois. Automatic linear correction of rounding errors. *BIT Numerical Mathematics*, 41(3) :515–539, 2001.

[50] Philippe Langlois. Analyse d'erreur en précision finie. In Alain Barraud, editor, *Outils d'analyse numérique pour l'Automatique*, Traité IC2, chapter 1, pages 19–52. Hermès Science, 2002.

[51] Philippe Langlois and Nicolas Louvet. Solving triangular systems more accurately and efficiently. In *Proceedings of the 17th IMACS World Congress, Paris*, volume CDROM, July 2005.

[52] Philippe Langlois and Nicolas Louvet. How to ensure a faithful polynomial evaluation with the compensated Horner algorithm. In *18th IEEE International Symposium on Computer Arithmetic*, pages 141–149. IEEE, June 2007.

[53] Philippe Langlois and Nicolas Louvet. More instruction level parallelism explains the actual efficiency of compensated algorithms. Technical Report hal-00165020, DALI Research Team, HAL-CCSD, July 2007. (Submitted).

[54] Philippe Langlois and Nicolas Louvet. Operator dependant compensated algorithms. In *Proceedings of the 12th GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics, Duisburg, Germany*, volume CDROM, 2007. 10 pages.

[55] Chen Li, Sylvain Pion, and Chee-Keng Yap. Recent progress in exact geometric computation. *Journal of Logic and Algebraic Programming*, 64(1) :85–111, 2005.

[56] Xiaoye S. Li, James W. Demmel, David H. Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, William Kahan, Suh Y. Kang, Anil Kapur, Michael C. Martin, Brandon J. Thompson, Teresa Tung, and Daniel J. Yoo. A reference implementation for the Extended and Mixed precision BLAS standard. Available at `http://crd.lbl.gov/~xiaoye/`.

[57] Xiaoye S. Li, James W. Demmel, David H. Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, William Kahan, Suh Y. Kang, Anil Kapur, Michael C. Martin, Brandon J. Thompson, Teresa Tung, and Daniel J. Yoo. Design, implementation and testing of extended and mixed precision BLAS. *ACM Trans. Math. Software*, 28(2) :152–205, 2002.

[58] Seppo Linnainmaa. Software for doubled-precision floating-point computations. *ACM Trans. Math. Softw.*, 7(3) :272–283, 1981.

[59] Peter Linz. Accurate floating-point summation. *Commun. ACM*, 13(6) :361–362, 1970.

[60] Michael A. Malcolm. On accurate floating-point summation. *Commun. ACM*, 14(11) :731–736, 1971.

[61] Peter Markstein. *IA-64 and elementary functions : speed and precision*. Prentice Hall, 2000.

[62] John Michael McNamee. A comparison of methods for accurate summation. *SIGSAM Bull.*, 38(1) :1–7, 2004.

[63] Cleve B. Moler. Iterative refinement in floating point. *J. ACM*, 14(2) :316–321, 1967.

[64] Ole Møller. Note on quasi double-precision. *BIT*, 5(4) :251–255, 1965.

[65] Ole Møller. Quasi double-precision in floating point addition. *BIT*, 5(1) :37–50, 1965.

[66] Jean-Michel Muller. *Elementary functions : algorithms and implementation*. Birkhäuser, Cambridge, MA, USA ; Berlin, Germany ; Basel, Switzerland, second edition, 2006.

[67] A. Neumaier. Rundungsfehleranalyse Einiger Verfahren Zur Summation Endlicher Summen. (German) [Rounding error analysis of a method for summation of finite sums]. *Zeitschrift für Angewandte Mathematik und Mechanik*, 54 :39–51, 1974.

[68] Yves Nievergelt. Scalar fused multiply-add instructions produce floating-point matrix arithmetic provably accurate to the penultimate digit. *ACM Trans. Math. Softw.*, 29(1) :27–48, 2003.

[69] Yves Nievergelt. Analysis and applications of Priest's distillation. *ACM Trans. Math. Softw.*, 30(4) :402–433, 2004.

[70] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6) :1955–1988, 2005.

[71] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Verified solution of linear systems without directed rounding. Technical Report No. 2005-04, Advanced Research Institute for Science and Engineering, Waseda University, 2005.

[72] T. Ohta, T. Ogita, S. M. Rump, and S. Oishi. Numerical method for dense linear systems with arbitrarily ill-conditioned matrices. In *Proceedings of International Symposium on Nonlinear Theory and its Applications, Bruge, Belgium, October 18-21*, pages 745–748, 2005.

[73] Shin'Ichi Oishi and Siegfried M. Rump. Fast verification of solutions of matrix equations. *Numerische Mathematik*, 90(4) :755–773, February 2002.

[74] M. A. Overton. *Numerical Computing with IEEE Floating-Point Arithmetic*. SIAM, 2001.

[75] K. Ozaki, T. Ogita, S. Miyajima, S. Oishi, and S. M. Rump. Numerical method for dense linear systems with arbitrariliy ill-conditioned matrices. In *Proceedings of International Symposium on Nonlinear Theory and its Applications, Bruge, Belgium, October 18-21*, pages 749–752, 2005.

[76] K. Ozaki, T. Ogita, S. Miyajima, S. Oishi, and S. M. Rump. A method of obtaining verified solutions for linear systems suited for Java. *J. Comput. Appl. Math.*, 199(2) :337–344, 2006.

[77] J. M. Peña. On the multivariate Horner scheme. *SIAM Journal on Numerical Analysis*, 37(4) :1186–1197, 2000.

[78] Michèle Pichat. Correction d'une somme en arithmétique à virgule flottante. *Numerische Mathematik*, 19 :400–406, 1972.

[79] Michèle Pichat. *Contributions à l'étude des erreurs d'arrondi en arithmétique à virgule flottante*. PhD thesis, Université Scientifique et Médicale de Grenoble, Grenoble, France, 1976.

[80] Michèle Pichat and Jean Vignes. *Ingénierie du contrôle de la précision des calculs sur ordinateur*. Editions Technip, 1993.

[81] D. M. Priest. Algorithms for arbitrary precision floating point arithmetic. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic (Arith-10)*, pages 132–144, Grenoble, France, 1991. IEEE Computer Society Press, Los Alamitos, CA.

[82] Douglas M. Priest. *On Properties of Floating Point Arithmetics : Numerical Stability and the Cost of Accurate Computations*. PhD thesis, Berkeley, CA, USA, 1992.

[83] S. M. Rump. Algorithms for computing validated results. In J. Grabmeier, E. Kaltofen, and V. Weispfennig, editors, *Computer Algebra Handbook*, pages 110–112. Springer-Verlag, 2003.

[84] S. M. Rump. Computer-assisted proofs and self-validating methods. In B. Einarsson, editor, *Handbook on Acuracy and Reliability in Scientific Computation*, pages 195–240. Society for Industrial and Applied Mathematics (SIAM), 2005.

[85] S. M. Rump and S. Oishi. Super-fast validated solution of linear systems. *J. Comput. Appl. Math.*, 199(2) :199–206, 2006.

[86] Siegfried M. Rump, Takeshi Ogita, and Shin'ichi Oishi. Accurate floating-point summation. Technical report, Institute for Reliable Computing, Hamburg University of Technology, 2006.

[87] Jonathan R. Shewchuk. Robust adaptive floating-point geometric predicates. In *SCG '96 : Proceedings of the twelfth annual symposium on Computational geometry*, pages 141–150, New York, NY, USA, 1996. ACM Press.

[88] Jonathan R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. In *Discrete and Computational Geometry*, volume 18, pages 305–363, 1997.

[89] J. H. Wilkinson. *Rounding Errors in Algebraic Processes.* Prentice-Hall, Englewood Cliffs, NJ, USA, 1963.

[90] Yong-Kang Zhu and Wayne Hayes. Fast, guaranteed-accurate sums of many floating-point numbers. In Guillaume. Hanrot and Paul Zimmermann, editors, *RNC7 : Proceedings of the 7th Conference on Real Numbers and Computers*, pages 11–22, 2006.

[91] Yong-Kang Zhu, Jun-Hai Yong, and Guo-Qin Zheng. A new distillation algorithm for floating-point summation. *SIAM Journal on Scientific Computing*, 26(6) :2066–2078, 2005.

Annexes

Nous reportons dans ces annexes les résultats des mesures de performances effectuées pour cette thèse. Les abréviations indiquées ci-dessous ont été utilisées pour améliorer le formatage des tableaux.

Abrév.	Algorithme
H	Horner (algorithme 2.6)
CH	CompHorner (algorithme 4.3)
DDH	DDHorner (algorithme 3.27)
CH_{fma}	$CompHorner_{fma}$ (algorithme 6.3)
CHFMA	CompHornerFMA (algorithme 6.8)
CH_4	CompHorner4 (voir Section 8.7)
QDH	QDHorner (voir Section 8.7)
H_{mpfr4}	MPFRHorner4 (voir Section 8.7)

Toutes les mesures reportées ci-après sont exprimées en nombre de cycles. Ces nombres de cycles ont été mesurés à l'aide des compteurs de performances présents sur les différentes architectures considérées.

Environnements expérimentaux utilisés

Environnement	Détails
(I)	Intel Pentium 4, 3 GHz, GCC 4.1.2, fpu x87 `-std=c99 -march=pentium4 -mfpmath=387 -O3 -funroll-all-loops`
(II)	Intel Pentium 4, 3 GHz, GCC 4.1.2, fpu sse `-std=c99 -march=pentium4 -mfpmath=sse -O3 -funroll-all-loops`
(III)	Intel Pentium 4, 3 GHz, ICC 9.1, fpu x87 `-c99 -mtune=pentium4 -O3 -funroll-loops -mp1`
(IV)	Intel Pentium 4, 3 GHz, ICC 9.1, fpu sse `-c99 -mtune=pentium4 -msse2 -O3 -funroll-loops -fp-model source`
(V)	AMD Athlon 64 3200+, 2 GHz, GCC 4.1.2, fpu sse `-std=c99 -march=athlon64 -msse2 -mfpmath=sse -O3 -funroll-all-loops`
(VI)	Itanium 2, 1.5 GHz, GCC 4.1.1 `-std=c99 -mtune=itanium2 -O3 -funroll-all-loops`
(VII)	Itanium 2, 1.5 GHz, ICC 9.1 `-c99 -O3 -mp -IPF_fma`

Performances de CompHorner et DDHorner

Environnement I : Intel Pentium 4, 3.00GHz, `gcc 4.1.2`, unité flottante x87

	Nombre de cycles				Temps d'exécution normalisé			
deg	H	CH	DDH	CH_{bnd}	H	CH	DDH	CH_{bnd}
10	255	578	1485	1095	1.0	2.3	5.8	4.3
15	329	826	2160	1320	1.0	2.5	6.6	4.0
20	398	998	2828	1530	1.0	2.5	7.1	3.8
25	472	1237	3495	1755	1.0	2.6	7.4	3.7
30	532	1418	4155	2070	1.0	2.7	7.8	3.9
35	608	1635	4859	2235	1.0	2.7	8.0	3.7
40	675	1868	5528	2558	1.0	2.8	8.2	3.8
45	749	2078	6203	2686	1.0	2.8	8.3	3.6
50	818	2281	6870	2954	1.0	2.8	8.4	3.6
55	892	2490	7538	3150	1.0	2.8	8.5	3.5
60	959	2708	8213	3443	1.0	2.8	8.6	3.6
65	1028	2918	8880	3615	1.0	2.8	8.6	3.5
70	1095	3128	9547	3870	1.0	2.9	8.7	3.5
75	1169	3338	10223	4088	1.0	2.9	8.7	3.5
80	1237	3548	10889	4388	1.0	2.9	8.8	3.5
85	1313	3742	11558	4530	1.0	2.8	8.8	3.5
90	1379	3968	12233	4822	1.0	2.9	8.9	3.5
95	1448	4170	12900	5003	1.0	2.9	8.9	3.5
100	1515	4388	13567	5242	1.0	2.9	9.0	3.5
105	1590	4604	14234	5459	1.0	2.9	9.0	3.4
110	1657	4807	14909	5752	1.0	2.9	9.0	3.5
115	1733	5017	15578	5933	1.0	2.9	9.0	3.4
120	1800	5228	16253	6233	1.0	2.9	9.0	3.5
125	1868	5445	16920	6390	1.0	2.9	9.1	3.4
130	1934	5641	17587	6630	1.0	2.9	9.1	3.4
135	2010	5850	18255	6848	1.0	2.9	9.1	3.4
140	2078	6068	18930	7155	1.0	2.9	9.1	3.4
145	2183	6277	19598	7313	1.0	2.9	9.0	3.3
150	2219	6488	20273	7620	1.0	2.9	9.1	3.4
155	2287	6698	20940	7793	1.0	2.9	9.2	3.4
160	2355	6901	21608	8054	1.0	2.9	9.2	3.4
165	2468	7118	22275	8235	1.0	2.9	9.0	3.3
170	2535	7328	22949	8519	1.0	2.9	9.1	3.4
175	2602	7538	23618	8715	1.0	2.9	9.1	3.3
180	2670	7740	24293	8947	1.0	2.9	9.1	3.4
185	2745	7950	24960	9143	1.0	2.9	9.1	3.3
190	2813	8168	25620	9464	1.0	2.9	9.1	3.4
195	2879	8385	26295	9616	1.0	2.9	9.1	3.3
200	2955	8581	26970	9907	1.0	2.9	9.1	3.4
				moyennes		2.8	8.6	3.5

Environnement II : Intel Pentium 4, 3.00GHz, `gcc 4.1.2`, unité flottante SSE

	Nombre de cycles				Temps d'exécution normalisé			
deg	H	CH	DDH	CH_{bnd}	H	CH	DDH	CH_{bnd}
10	255	578	1388	1186	1.0	2.3	5.4	4.7
15	308	780	1980	1358	1.0	2.5	6.4	4.4
20	368	1012	2588	1756	1.0	2.8	7.0	4.8
25	428	1170	3203	1778	1.0	2.7	7.5	4.2
30	488	1403	3810	2010	1.0	2.9	7.8	4.1
35	555	1568	4418	2212	1.0	2.8	8.0	4.0
40	608	1800	5033	2423	1.0	3.0	8.3	4.0
45	668	1988	5678	2647	1.0	3.0	8.5	4.0
50	735	2220	6284	3135	1.0	3.0	8.5	4.3
55	788	2378	6900	3075	1.0	3.0	8.8	3.9
60	848	2609	7508	3285	1.0	3.1	8.9	3.9
65	908	2775	8115	3503	1.0	3.1	8.9	3.9
70	968	3008	8723	3721	1.0	3.1	9.0	3.8
75	1035	3173	9338	3938	1.0	3.1	9.0	3.8
80	1088	3405	9945	4163	1.0	3.1	9.1	3.8
85	1148	3570	10560	4381	1.0	3.1	9.2	3.8
90	1215	3781	11168	4636	1.0	3.1	9.2	3.8
95	1268	3960	11775	4793	1.0	3.1	9.3	3.8
100	1328	4193	12383	5010	1.0	3.2	9.3	3.8
105	1388	4358	12998	5228	1.0	3.1	9.4	3.8
110	1448	4575	13605	5445	1.0	3.2	9.4	3.8
115	1515	4747	14220	5663	1.0	3.1	9.4	3.7
120	1568	4951	14828	5888	1.0	3.2	9.5	3.8
125	1628	5153	15435	6083	1.0	3.2	9.5	3.7
130	1695	5370	16043	6300	1.0	3.2	9.5	3.7
135	1747	5543	16658	6548	1.0	3.2	9.5	3.7
140	1808	5775	17265	6743	1.0	3.2	9.5	3.7
145	1889	5940	17880	6968	1.0	3.1	9.5	3.7
150	1928	6135	18488	7193	1.0	3.2	9.6	3.7
155	1995	6330	19095	7380	1.0	3.2	9.6	3.7
160	2055	6532	19703	7606	1.0	3.2	9.6	3.7
165	2129	6727	20318	7808	1.0	3.2	9.5	3.7
170	2190	6960	20925	8025	1.0	3.2	9.6	3.7
175	2250	7118	21540	8243	1.0	3.2	9.6	3.7
180	2310	7350	22148	8483	1.0	3.2	9.6	3.7
185	2370	7515	22755	8670	1.0	3.2	9.6	3.7
190	2430	7718	23363	8895	1.0	3.2	9.6	3.7
195	2490	7913	23978	9106	1.0	3.2	9.6	3.7
200	2549	8145	24585	9345	1.0	3.2	9.6	3.7
				moyennes		3.1	8.9	3.9

Environnement III : Intel Pentium 4, 3.00GHz, `icc 9.1`, unité flottante x87

	Nombre de cycles				Temps d'exécution normalisé			
deg	H	CH	DDH	CH_{bnd}	H	CH	DDH	CH_{bnd}
10	135	480	1350	608	1.0	3.6	10.0	4.5
15	202	682	2025	833	1.0	3.4	10.0	4.1
20	368	908	2806	1080	1.0	2.5	7.6	2.9
25	442	1118	3457	1305	1.0	2.5	7.8	3.0
30	511	1313	4125	1545	1.0	2.6	8.1	3.0
35	585	1463	4800	1778	1.0	2.5	8.2	3.0
40	653	1665	5483	2018	1.0	2.5	8.4	3.1
45	719	1845	6135	2242	1.0	2.6	8.5	3.1
50	795	2063	6825	2460	1.0	2.6	8.6	3.1
55	862	2235	7500	2707	1.0	2.6	8.7	3.1
60	922	2430	8137	2910	1.0	2.6	8.8	3.2
65	997	2602	8820	3158	1.0	2.6	8.8	3.2
70	1065	2798	9480	3398	1.0	2.6	8.9	3.2
75	1147	2978	10148	3630	1.0	2.6	8.8	3.2
80	1208	3211	10838	3863	1.0	2.7	9.0	3.2
85	1275	3359	11513	4126	1.0	2.6	9.0	3.2
90	1351	3562	12172	4320	1.0	2.6	9.0	3.2
95	1424	3757	12848	4553	1.0	2.6	9.0	3.2
100	1485	3952	13507	4763	1.0	2.7	9.1	3.2
105	1568	4139	14189	4980	1.0	2.6	9.0	3.2
110	1634	4313	14858	5213	1.0	2.6	9.1	3.2
115	1702	4500	15510	5461	1.0	2.6	9.1	3.2
120	1770	4703	16178	5737	1.0	2.7	9.1	3.2
125	1846	4898	16867	5932	1.0	2.7	9.1	3.2
130	1928	5115	17536	6142	1.0	2.7	9.1	3.2
135	1965	5257	18203	6397	1.0	2.7	9.3	3.3
140	2054	5453	18870	6599	1.0	2.7	9.2	3.2
145	2115	5655	19544	6825	1.0	2.7	9.2	3.2
150	2190	5881	20197	7088	1.0	2.7	9.2	3.2
155	2250	6068	20895	7320	1.0	2.7	9.3	3.3
160	2333	6225	21540	7553	1.0	2.7	9.2	3.2
165	2393	6458	22215	7747	1.0	2.7	9.3	3.2
170	2467	6608	22897	7980	1.0	2.7	9.3	3.2
175	2542	6795	23566	8250	1.0	2.7	9.3	3.2
180	2609	6982	24217	8438	1.0	2.7	9.3	3.2
185	2677	7193	24901	8723	1.0	2.7	9.3	3.3
190	2760	7357	25575	8940	1.0	2.7	9.3	3.2
195	2820	7552	26227	9127	1.0	2.7	9.3	3.2
200	2888	7740	26909	9412	1.0	2.7	9.3	3.3
				moyennes		2.7	9.0	3.2

Environnement IV : Intel Pentium 4, 3.00GHz, `icc 9.1`, unité flottante SSE

	Nombre de cycles				Temps d'exécution normalisé			
deg	H	CH	DDH	CH_{bnd}	H	CH	DDH	CH_{bnd}
10	120	479	1312	630	1.0	4.0	10.9	5.2
15	194	712	1943	878	1.0	3.7	10.0	4.5
20	352	937	2663	1148	1.0	2.7	7.6	3.3
25	397	1155	3292	1395	1.0	2.9	8.3	3.5
30	450	1350	3930	1634	1.0	3.0	8.7	3.6
35	525	1559	4560	1882	1.0	3.0	8.7	3.6
40	577	1763	5205	2130	1.0	3.1	9.0	3.7
45	630	1972	5850	2362	1.0	3.1	9.3	3.7
50	690	2212	6473	2617	1.0	3.2	9.4	3.8
55	742	2377	7110	2858	1.0	3.2	9.6	3.9
60	818	2588	7732	3105	1.0	3.2	9.5	3.8
65	870	2798	8408	3352	1.0	3.2	9.7	3.9
70	930	2992	9015	3600	1.0	3.2	9.7	3.9
75	990	3210	9645	3840	1.0	3.2	9.7	3.9
80	1050	3405	10283	4088	1.0	3.2	9.8	3.9
85	1117	3614	10920	4334	1.0	3.2	9.8	3.9
90	1163	3825	11550	4583	1.0	3.3	9.9	3.9
95	1238	4020	12188	4822	1.0	3.2	9.8	3.9
100	1290	4230	12839	5063	1.0	3.3	10.0	3.9
105	1349	4432	13493	5317	1.0	3.3	10.0	3.9
110	1417	4643	14085	5557	1.0	3.3	9.9	3.9
115	1477	4845	14730	5798	1.0	3.3	10.0	3.9
120	1538	5047	15360	6030	1.0	3.3	10.0	3.9
125	1598	5251	16050	6285	1.0	3.3	10.0	3.9
130	1657	5460	16627	6540	1.0	3.3	10.0	3.9
135	1710	5669	17273	6781	1.0	3.3	10.1	4.0
140	1785	5873	17903	7028	1.0	3.3	10.0	3.9
145	1830	6067	18548	7267	1.0	3.3	10.1	4.0
150	1898	6277	19170	7516	1.0	3.3	10.1	4.0
155	1964	6473	19800	7763	1.0	3.3	10.1	4.0
160	2025	6675	20444	8010	1.0	3.3	10.1	4.0
165	2069	6893	21083	8258	1.0	3.3	10.2	4.0
170	2145	7095	21720	8490	1.0	3.3	10.1	4.0
175	2197	7304	22357	8730	1.0	3.3	10.2	4.0
180	2257	7508	22988	8986	1.0	3.3	10.2	4.0
185	2310	7709	23618	9225	1.0	3.3	10.2	4.0
190	2369	7919	24254	9473	1.0	3.3	10.2	4.0
195	2438	8123	24885	9712	1.0	3.3	10.2	4.0
200	2497	8325	25522	9967	1.0	3.3	10.2	4.0
				moyennes		3.3	9.8	3.9

Environnement V : AMD Athlon 64 3200+, 2GHz, gcc 4.1.2, unité flottante SSE

	Nombre de cycles				Temps d'exécution normalisé			
deg	H	CH	DDH	CH_{bnd}	H	CH	DDH	CH_{bnd}
10	137	343	831	433	1.0	2.5	6.1	3.2
15	180	475	1215	578	1.0	2.6	6.8	3.2
20	219	631	1598	726	1.0	2.9	7.3	3.3
25	261	767	1995	912	1.0	2.9	7.6	3.5
30	299	901	2379	1059	1.0	3.0	8.0	3.5
35	337	1033	2763	1205	1.0	3.1	8.2	3.6
40	379	1180	3146	1353	1.0	3.1	8.3	3.6
45	419	1312	3530	1502	1.0	3.1	8.4	3.6
50	458	1446	3914	1649	1.0	3.2	8.5	3.6
55	498	1578	4298	1795	1.0	3.2	8.6	3.6
60	541	1725	4681	1943	1.0	3.2	8.7	3.6
65	579	1857	5065	2092	1.0	3.2	8.7	3.6
70	621	1991	5449	2240	1.0	3.2	8.8	3.6
75	668	2123	5833	2391	1.0	3.2	8.7	3.6
80	698	2270	6216	2533	1.0	3.3	8.9	3.6
85	752	2402	6600	2682	1.0	3.2	8.8	3.6
90	794	2536	6984	2829	1.0	3.2	8.8	3.6
95	834	2668	7368	2975	1.0	3.2	8.8	3.6
100	871	2815	7751	3123	1.0	3.2	8.9	3.6
105	912	2947	8135	3272	1.0	3.2	8.9	3.6
110	951	3081	8519	3419	1.0	3.2	9.0	3.6
115	991	3213	8903	3565	1.0	3.2	9.0	3.6
120	1034	3369	9286	3713	1.0	3.3	9.0	3.6
125	1074	3492	9670	3862	1.0	3.3	9.0	3.6
130	1111	3626	10054	4009	1.0	3.3	9.0	3.6
135	1151	3758	10438	4155	1.0	3.3	9.1	3.6
140	1192	3905	10821	4303	1.0	3.3	9.1	3.6
145	1234	4037	11205	4452	1.0	3.3	9.1	3.6
150	1272	4171	11589	4599	1.0	3.3	9.1	3.6
155	1312	4303	11973	4745	1.0	3.3	9.1	3.6
160	1351	4450	12356	4893	1.0	3.3	9.1	3.6
165	1391	4582	12740	5042	1.0	3.3	9.2	3.6
170	1431	4716	13124	5189	1.0	3.3	9.2	3.6
175	1471	4848	13508	5335	1.0	3.3	9.2	3.6
180	1514	4995	13891	5483	1.0	3.3	9.2	3.6
185	1551	5127	14275	5632	1.0	3.3	9.2	3.6
190	1594	5261	14659	5779	1.0	3.3	9.2	3.6
195	1634	5393	15043	5925	1.0	3.3	9.2	3.6
200	1671	5540	15426	6073	1.0	3.3	9.2	3.6
				moyennes		3.2	8.7	3.6

Environnement VI : Intel Itanium 2, 1.5 GHz, `gcc 4.1.1`

	Nombre de cycles				Temps d'exécution normalisé			
deg	H	CH	DDH	CH_{bnd}	H	CH	DDH	CH_{bnd}
10	191	345	607	487	1.0	1.8	3.2	2.5
15	224	449	840	597	1.0	2.0	3.8	2.7
20	256	552	1077	714	1.0	2.2	4.2	2.8
25	269	601	1310	770	1.0	2.2	4.9	2.9
30	303	710	1547	889	1.0	2.3	5.1	2.9
35	327	814	1780	1007	1.0	2.5	5.4	3.1
40	359	917	2017	1124	1.0	2.6	5.6	3.1
45	383	970	2250	1180	1.0	2.5	5.9	3.1
50	406	1075	2487	1299	1.0	2.6	6.1	3.2
55	439	1179	2720	1417	1.0	2.7	6.2	3.2
60	463	1282	2957	1534	1.0	2.8	6.4	3.3
65	484	1335	3190	1590	1.0	2.8	6.6	3.3
70	518	1440	3427	1709	1.0	2.8	6.6	3.3
75	542	1544	3660	1827	1.0	2.8	6.8	3.4
80	574	1647	3897	1944	1.0	2.9	6.8	3.4
85	598	1700	4130	1999	1.0	2.8	6.9	3.3
90	621	1805	4367	2117	1.0	2.9	7.0	3.4
95	654	1909	4600	2237	1.0	2.9	7.0	3.4
100	678	2012	4837	2354	1.0	3.0	7.1	3.5
105	699	2065	5070	2410	1.0	3.0	7.3	3.4
110	733	2170	5307	2527	1.0	3.0	7.2	3.4
115	757	2274	5540	2645	1.0	3.0	7.3	3.5
120	789	2377	5777	2764	1.0	3.0	7.3	3.5
125	813	2430	6010	2820	1.0	3.0	7.4	3.5
130	834	2535	6247	2937	1.0	3.0	7.5	3.5
135	869	2639	6480	3055	1.0	3.0	7.5	3.5
140	890	2742	6717	3174	1.0	3.1	7.5	3.6
145	914	2795	6950	3230	1.0	3.1	7.6	3.5
150	948	2900	7187	3347	1.0	3.1	7.6	3.5
155	972	3004	7420	3465	1.0	3.1	7.6	3.6
160	1004	3107	7657	3584	1.0	3.1	7.6	3.6
165	1028	3160	7890	3640	1.0	3.1	7.7	3.5
170	1051	3265	8127	3757	1.0	3.1	7.7	3.6
175	1084	3369	8360	3877	1.0	3.1	7.7	3.6
180	1108	3472	8597	3994	1.0	3.1	7.8	3.6
185	1129	3525	8830	4048	1.0	3.1	7.8	3.6
190	1163	3630	9067	4167	1.0	3.1	7.8	3.6
195	1187	3734	9300	4285	1.0	3.1	7.8	3.6
200	1219	3837	9537	4404	1.0	3.1	7.8	3.6
				moyennes		2.8	6.7	3.4

Environnement VII : Intel Itanium 2, 1.5 GHz, `icc 9.1`

	Nombre de cycles				Temps d'exécution normalisé			
deg	H	CH	DDH	CH_{bnd}	H	CH	DDH	CH_{bnd}
10	386	448	793	562	1.0	1.2	2.1	1.5
15	406	488	1013	602	1.0	1.2	2.5	1.5
20	434	528	1233	642	1.0	1.2	2.8	1.5
25	457	568	1453	682	1.0	1.2	3.2	1.5
30	474	608	1673	722	1.0	1.3	3.5	1.5
35	506	648	1893	762	1.0	1.3	3.7	1.5
40	514	688	2113	802	1.0	1.3	4.1	1.6
45	537	728	2333	842	1.0	1.4	4.3	1.6
50	554	768	2553	882	1.0	1.4	4.6	1.6
55	577	808	2773	922	1.0	1.4	4.8	1.6
60	594	850	2995	962	1.0	1.4	5.0	1.6
65	617	888	3213	1002	1.0	1.4	5.2	1.6
70	634	928	3433	1042	1.0	1.5	5.4	1.6
75	666	968	3653	1082	1.0	1.5	5.5	1.6
80	674	1008	3873	1122	1.0	1.5	5.7	1.7
85	697	1048	4093	1162	1.0	1.5	5.9	1.7
90	714	1088	4313	1202	1.0	1.5	6.0	1.7
95	737	1128	4533	1242	1.0	1.5	6.2	1.7
100	754	1168	4753	1282	1.0	1.5	6.3	1.7
105	777	1208	4973	1322	1.0	1.6	6.4	1.7
110	794	1248	5193	1362	1.0	1.6	6.5	1.7
115	826	1288	5413	1402	1.0	1.6	6.6	1.7
120	834	1328	5633	1442	1.0	1.6	6.8	1.7
125	857	1370	5855	1482	1.0	1.6	6.8	1.7
130	874	1408	6073	1522	1.0	1.6	6.9	1.7
135	897	1448	6293	1562	1.0	1.6	7.0	1.7
140	914	1488	6513	1602	1.0	1.6	7.1	1.8
145	937	1528	6733	1642	1.0	1.6	7.2	1.8
150	954	1568	6953	1682	1.0	1.6	7.3	1.8
155	986	1608	7173	1722	1.0	1.6	7.3	1.7
160	994	1648	7393	1762	1.0	1.7	7.4	1.8
165	1017	1688	7613	1802	1.0	1.7	7.5	1.8
170	1034	1728	7833	1842	1.0	1.7	7.6	1.8
175	1057	1768	8053	1882	1.0	1.7	7.6	1.8
180	1074	1808	8273	1922	1.0	1.7	7.7	1.8
185	1097	1848	8493	1962	1.0	1.7	7.7	1.8
190	1114	1888	8713	2002	1.0	1.7	7.8	1.8
195	1146	1928	8933	2042	1.0	1.7	7.8	1.8
200	1154	1968	9153	2082	1.0	1.7	7.9	1.8
				moyennes		1.5	5.9	1.7

Performances de CompHorner en présence d'un FMA

Environnement VI : Intel Itanium 2, 1.5 GHz, `gcc 4.1.1`

	Nombre de cycles				Temps d'exécution normalisé			
deg	H	CH	CH_{fma}	DDH	H	CH	CH_{fma}	DDH
10	191	345	513	607	1.0	1.8	2.7	3.2
15	224	449	684	840	1.0	2.0	3.1	3.8
20	256	552	873	1077	1.0	2.2	3.4	4.2
25	269	601	1044	1310	1.0	2.2	3.9	4.9
30	303	710	1233	1547	1.0	2.3	4.1	5.1
35	327	814	1404	1780	1.0	2.5	4.3	5.4
40	359	917	1593	2017	1.0	2.6	4.4	5.6
45	383	970	1764	2250	1.0	2.5	4.6	5.9
50	406	1075	1953	2487	1.0	2.6	4.8	6.1
55	439	1179	2124	2720	1.0	2.7	4.8	6.2
60	463	1282	2313	2957	1.0	2.8	5.0	6.4
65	484	1335	2484	3190	1.0	2.8	5.1	6.6
70	518	1440	2673	3427	1.0	2.8	5.2	6.6
75	542	1544	2844	3660	1.0	2.8	5.2	6.8
80	574	1647	3033	3897	1.0	2.9	5.3	6.8
85	598	1700	3204	4130	1.0	2.8	5.4	6.9
90	621	1805	3393	4367	1.0	2.9	5.5	7.0
95	654	1909	3564	4600	1.0	2.9	5.4	7.0
100	678	2012	3753	4837	1.0	3.0	5.5	7.1
105	699	2065	3924	5070	1.0	3.0	5.6	7.3
110	733	2170	4113	5307	1.0	3.0	5.6	7.2
115	757	2274	4284	5540	1.0	3.0	5.7	7.3
120	789	2377	4473	5777	1.0	3.0	5.7	7.3
125	813	2430	4644	6010	1.0	3.0	5.7	7.4
130	834	2535	4833	6247	1.0	3.0	5.8	7.5
135	869	2639	5004	6480	1.0	3.0	5.8	7.5
140	890	2742	5193	6717	1.0	3.1	5.8	7.5
145	914	2795	5364	6950	1.0	3.1	5.9	7.6
150	948	2900	5553	7187	1.0	3.1	5.9	7.6
155	972	3004	5724	7420	1.0	3.1	5.9	7.6
160	1004	3107	5913	7657	1.0	3.1	5.9	7.6
165	1028	3160	6084	7890	1.0	3.1	5.9	7.7
170	1051	3265	6273	8127	1.0	3.1	6.0	7.7
175	1084	3369	6444	8360	1.0	3.1	5.9	7.7
180	1108	3472	6633	8597	1.0	3.1	6.0	7.8
185	1129	3525	6804	8830	1.0	3.1	6.0	7.8
190	1163	3630	6993	9067	1.0	3.1	6.0	7.8
195	1187	3734	7164	9300	1.0	3.1	6.0	7.8
200	1219	3837	7353	9537	1.0	3.1	6.0	7.8
				moyennes		2.8	5.3	6.7

Environnement VII : Intel Itanium 2, 1.5 GHz, `icc 9.1`

	Nombre de cycles				Temps d'exécution normalisé			
deg	H	CH	CH_{fma}	DDH	H	CH	CH_{fma}	DDH
10	386	448	480	793	1.0	1.2	1.2	2.1
15	406	488	530	1013	1.0	1.2	1.3	2.5
20	434	528	580	1233	1.0	1.2	1.3	2.8
25	457	568	630	1453	1.0	1.2	1.4	3.2
30	474	608	680	1673	1.0	1.3	1.4	3.5
35	506	648	730	1893	1.0	1.3	1.4	3.7
40	514	688	780	2113	1.0	1.3	1.5	4.1
45	537	728	830	2333	1.0	1.4	1.5	4.3
50	554	768	880	2553	1.0	1.4	1.6	4.6
55	577	808	930	2773	1.0	1.4	1.6	4.8
60	594	850	982	2995	1.0	1.4	1.7	5.0
65	617	888	1030	3213	1.0	1.4	1.7	5.2
70	634	928	1080	3433	1.0	1.5	1.7	5.4
75	666	968	1130	3653	1.0	1.5	1.7	5.5
80	674	1008	1180	3873	1.0	1.5	1.8	5.7
85	697	1048	1230	4093	1.0	1.5	1.8	5.9
90	714	1088	1280	4313	1.0	1.5	1.8	6.0
95	737	1128	1330	4533	1.0	1.5	1.8	6.2
100	754	1168	1380	4753	1.0	1.5	1.8	6.3
105	777	1208	1430	4973	1.0	1.6	1.8	6.4
110	794	1248	1480	5193	1.0	1.6	1.9	6.5
115	826	1288	1530	5413	1.0	1.6	1.9	6.6
120	834	1328	1580	5633	1.0	1.6	1.9	6.8
125	857	1370	1632	5855	1.0	1.6	1.9	6.8
130	874	1408	1680	6073	1.0	1.6	1.9	6.9
135	897	1448	1730	6293	1.0	1.6	1.9	7.0
140	914	1488	1780	6513	1.0	1.6	1.9	7.1
145	937	1528	1830	6733	1.0	1.6	2.0	7.2
150	954	1568	1880	6953	1.0	1.6	2.0	7.3
155	986	1608	1930	7173	1.0	1.6	2.0	7.3
160	994	1648	1980	7393	1.0	1.7	2.0	7.4
165	1017	1688	2030	7613	1.0	1.7	2.0	7.5
170	1034	1728	2080	7833	1.0	1.7	2.0	7.6
175	1057	1768	2130	8053	1.0	1.7	2.0	7.6
180	1074	1808	2180	8273	1.0	1.7	2.0	7.7
185	1097	1848	2230	8493	1.0	1.7	2.0	7.7
190	1114	1888	2280	8713	1.0	1.7	2.0	7.8
195	1146	1928	2330	8933	1.0	1.7	2.0	7.8
200	1154	1968	2380	9153	1.0	1.7	2.1	7.9
				moyennes		1.5	1.8	5.9

Performances de CompHornerK et de CompHornerKBound

Environnement I : Intel Pentium 4, 3.00GHz, `gcc 4.1.2`, unité flottante x87

K	CompHornerK/Horner	MPFRHornerK/Horner	CompHornerKBound/CompHornerK
2	5.5	48.7	1.37
3	12.8	53.4	1.28
4	27.5	58.0	1.13
5	53.3	57.1	1.31
6	121.7	62.4	1.19
7	255.1	65.8	1.17

Environnement III : Intel Pentium 4, 3.00GHz, `icc 9.1`, unité flottante x87

K	CompHornerK/Horner	MPFRHornerK/Horner	CompHornerKBound/CompHornerK
2	7.1	69.6	1.21
3	18.1	74.1	1.04
4	36.0	77.2	1.03
5	69.7	80.1	1.14
6	149.4	84.5	1.05
7	303.6	91.3	1.05

Environnement V : AMD Athlon 64 3200+, 2GHz, gcc 4.1.2, unité flottante SSE

K	CompHornerK/Horner	MPFRHornerK/Horner	CompHornerKBound/CompHornerK
2	4.4	29.1	1.19
3	10.7	35.6	1.17
4	22.9	43.4	1.09
5	48.4	47.6	1.09
6	104.8	46.6	1.07
7	208.4	60.3	1.09

Environnement VI : Intel Itanium 2, 1.5 GHz, `gcc 4.1.1`

K	CompHornerK/Horner	MPFRHornerK/Horner	CompHornerKBound/CompHornerK
2	4.1	47.8	1.66
3	22.4	55.8	1.21
4	50.3	59.8	1.18
5	107.2	61.6	1.16
6	219.8	64.3	1.15
7	447.4	64.6	1.15

Environnement VII : Intel Itanium 2, 1.5 GHz, `icc 9.1`

K	CompHornerK/Horner	MPFRHornerK/Horner	CompHornerKBound/CompHornerK
2	1.8	42.1	1.21
3	17.7	49.1	1.07
4	26.7	52.2	1.07
5	46.0	54.1	1.06
6	86.9	56.5	1.06
7	173.4	56.4	1.05

Performances de CompHorner4

Environnement I : Intel Pentium 4, 3.00GHz, `gcc 4.1.2`, unité flottante x87

	Nombre de cycles				Temps d'exécution normalisé			
deg	H	CH_4	QDH	H_{mpfr4}	H	CH_4	QDH	H_{mpfr4}
10	255	3960	5384	8909	1.0	15.5	21.1	34.9
15	329	5393	7920	12540	1.0	16.4	24.1	38.1
20	398	6960	10470	15435	1.0	17.5	26.3	38.8
25	472	8573	13034	20986	1.0	18.2	27.6	44.5
30	532	10073	15608	24375	1.0	18.9	29.3	45.8
35	608	11610	18142	26730	1.0	19.1	29.8	44.0
40	675	13140	20715	32205	1.0	19.5	30.7	47.7
45	749	14715	23250	35063	1.0	19.6	31.0	46.8
50	818	16141	25823	38708	1.0	19.7	31.6	47.3
55	892	17768	28380	44393	1.0	19.9	31.8	49.8
60	959	19253	30929	46853	1.0	20.1	32.3	48.9
65	1028	20782	33488	47175	1.0	20.2	32.6	45.9
70	1095	22274	36045	53010	1.0	20.3	32.9	48.4
75	1169	23835	38603	59588	1.0	20.4	33.0	51.0
80	1237	25418	41153	61792	1.0	20.5	33.3	50.0
85	1313	26903	43710	66623	1.0	20.5	33.3	50.7
90	1379	28380	46268	72044	1.0	20.6	33.6	52.2
95	1448	29955	48802	70770	1.0	20.7	33.7	48.9
100	1515	31493	51374	74782	1.0	20.8	33.9	49.4
105	1590	33030	53932	81660	1.0	20.8	33.9	51.4
110	1657	34576	56482	79988	1.0	20.9	34.1	48.3
115	1733	36083	59039	88800	1.0	20.8	34.1	51.2
120	1800	37538	61598	88455	1.0	20.9	34.2	49.1
125	1868	39120	64148	92528	1.0	20.9	34.3	49.5
130	1934	40635	66698	102151	1.0	21.0	34.5	52.8
135	2010	42181	69240	104114	1.0	21.0	34.4	51.8
140	2078	43711	71798	108683	1.0	21.0	34.6	52.3
145	2183	45278	74362	109469	1.0	20.7	34.1	50.1
150	2219	46755	76928	116580	1.0	21.1	34.7	52.5
155	2287	48360	79462	115349	1.0	21.1	34.7	50.4
160	2355	49845	82034	124665	1.0	21.2	34.8	52.9
165	2468	51330	84584	129781	1.0	20.8	34.3	52.6
170	2535	52898	87143	126720	1.0	20.9	34.4	50.0
175	2602	54480	89692	128850	1.0	20.9	34.5	49.5
180	2670	55973	92250	133028	1.0	21.0	34.6	49.8
185	2745	57428	94785	136522	1.0	20.9	34.5	49.7
190	2813	59033	97342	142223	1.0	21.0	34.6	50.6
195	2879	60466	99899	150293	1.0	21.0	34.7	52.2
200	2955	62093	102473	147944	1.0	21.0	34.7	50.1
				moyennes		20.2	32.6	48.7

Environnement II : Intel Pentium 4, 3.00GHz, `gcc 4.1.2`, unité flottante SSE

	Nombre de cycles				Temps d'exécution normalisé			
deg	H	CH_4	QDH	H_{mpfr4}	H	CH_4	QDH	H_{mpfr4}
10	255	4342	5355	8738	1.0	17.0	21.0	34.3
15	308	5580	7830	13305	1.0	18.1	25.4	43.2
20	368	6946	10373	15990	1.0	18.9	28.2	43.5
25	428	8513	12848	20520	1.0	19.9	30.0	47.9
30	488	9885	15353	24113	1.0	20.3	31.5	49.4
35	555	11498	17873	29685	1.0	20.7	32.2	53.5
40	608	12840	20378	31710	1.0	21.1	33.5	52.2
45	668	14483	22890	35715	1.0	21.7	34.3	53.5
50	735	15848	25433	40012	1.0	21.6	34.6	54.4
55	788	17437	27938	42060	1.0	22.1	35.5	53.4
60	848	18840	30413	43995	1.0	22.2	35.9	51.9
65	908	20408	32933	50205	1.0	22.5	36.3	55.3
70	968	21818	35460	53933	1.0	22.5	36.6	55.7
75	1035	23325	37950	60593	1.0	22.5	36.7	58.5
80	1088	24788	40463	62100	1.0	22.8	37.2	57.1
85	1148	26295	42960	67673	1.0	22.9	37.4	58.9
90	1215	27818	45503	70635	1.0	22.9	37.5	58.1
95	1268	29333	47993	74775	1.0	23.1	37.8	59.0
100	1328	30878	50490	75435	1.0	23.3	38.0	56.8
105	1388	32340	53003	77828	1.0	23.3	38.2	56.1
110	1448	33878	55523	84931	1.0	23.4	38.3	58.7
115	1515	35243	58043	94673	1.0	23.3	38.3	62.5
120	1568	36735	60540	92783	1.0	23.4	38.6	59.2
125	1628	38370	63038	95326	1.0	23.6	38.7	58.6
130	1695	39833	65565	102195	1.0	23.5	38.7	60.3
135	1747	41265	68063	100411	1.0	23.6	39.0	57.5
140	1808	42780	70575	107693	1.0	23.7	39.0	59.6
145	1889	44198	73080	102143	1.0	23.4	38.7	54.1
150	1928	45743	75623	118710	1.0	23.7	39.2	61.6
155	1995	47198	78105	120885	1.0	23.7	39.2	60.6
160	2055	48705	80618	120375	1.0	23.7	39.2	58.6
165	2129	50273	83130	120698	1.0	23.6	39.0	56.7
170	2190	51675	85628	123548	1.0	23.6	39.1	56.4
175	2250	53198	88140	132803	1.0	23.6	39.2	59.0
180	2310	54630	90660	136680	1.0	23.6	39.2	59.2
185	2370	56228	93173	143566	1.0	23.7	39.3	60.6
190	2430	57623	95670	146438	1.0	23.7	39.4	60.3
195	2490	59189	98213	148485	1.0	23.8	39.4	59.6
200	2549	60638	100695	149910	1.0	23.8	39.5	58.8
				moyennes		22.5	36.4	55.7

Environnement III : Intel Pentium 4, 3.00GHz, `icc 9.1`, unité flottante x87

	Nombre de cycles				Temps d'exécution normalisé			
deg	H	CH_4	QDH	H_{mpfr4}	H	CH_4	QDH	H_{mpfr4}
10	135	3975	6795	8910	1.0	29.4	50.3	66.0
15	202	5475	10103	12615	1.0	27.1	50.0	62.5
20	368	6953	13380	15818	1.0	18.9	36.4	43.0
25	442	8520	16710	19987	1.0	19.3	37.8	45.2
30	511	10034	20026	23632	1.0	19.6	39.2	46.2
35	585	11498	23294	27907	1.0	19.7	39.8	47.7
40	653	12998	26588	32070	1.0	19.9	40.7	49.1
45	719	14483	29902	36653	1.0	20.1	41.6	51.0
50	795	16027	33195	39510	1.0	20.2	41.8	49.7
55	862	17535	36480	42990	1.0	20.3	42.3	49.9
60	922	19027	39817	46875	1.0	20.6	43.2	50.8
65	997	20497	43080	49057	1.0	20.6	43.2	49.2
70	1065	21998	46387	53731	1.0	20.7	43.6	50.5
75	1147	23498	49703	57149	1.0	20.5	43.3	49.8
80	1208	25027	52995	60367	1.0	20.7	43.9	50.0
85	1275	26543	56295	63067	1.0	20.8	44.2	49.5
90	1351	28004	59580	66975	1.0	20.7	44.1	49.6
95	1424	29497	62903	70695	1.0	20.7	44.2	49.6
100	1485	31020	66187	79171	1.0	20.9	44.6	53.3
105	1568	32528	69495	78120	1.0	20.7	44.3	49.8
110	1634	33998	72795	84727	1.0	20.8	44.6	51.9
115	1702	35491	76087	88012	1.0	20.9	44.7	51.7
120	1770	37019	79403	90524	1.0	20.9	44.9	51.1
125	1846	38520	82687	98543	1.0	20.9	44.8	53.4
130	1928	40035	85995	107085	1.0	20.8	44.6	55.5
135	1965	41512	89288	100740	1.0	21.1	45.4	51.3
140	2054	43021	92609	112132	1.0	20.9	45.1	54.6
145	2115	44520	95880	110633	1.0	21.0	45.3	52.3
150	2190	46027	99188	113085	1.0	21.0	45.3	51.6
155	2250	47512	102487	118013	1.0	21.1	45.5	52.5
160	2333	49013	105780	120120	1.0	21.0	45.3	51.5
165	2393	50520	109080	123203	1.0	21.1	45.6	51.5
170	2467	52019	112387	130507	1.0	21.1	45.6	52.9
175	2542	53513	115688	131175	1.0	21.1	45.5	51.6
180	2609	54952	118980	138465	1.0	21.1	45.6	53.1
185	2677	56551	122288	135719	1.0	21.1	45.7	50.7
190	2760	58043	125588	146107	1.0	21.0	45.5	52.9
195	2820	59528	128895	152003	1.0	21.1	45.7	53.9
200	2888	61028	132209	159240	1.0	21.1	45.8	55.1
				moyennes		21.0	44.1	51.6

Environnement IV : Intel Pentium 4, 3.00GHz, `icc 9.1`, unité flottante SSE

	Nombre de cycles				Temps d'exécution normalisé			
deg	H	CH_4	QDH	H_{mpfr4}	H	CH_4	QDH	H_{mpfr4}
10	120	3623	5378	8760	1.0	30.2	44.8	73.0
15	194	5085	7942	11849	1.0	26.2	40.9	61.1
20	352	6518	10522	16447	1.0	18.5	29.9	46.7
25	397	8040	13095	20078	1.0	20.3	33.0	50.6
30	450	9405	15659	23257	1.0	20.9	34.8	51.7
35	525	10852	18254	27742	1.0	20.7	34.8	52.8
40	577	12450	20812	31732	1.0	21.6	36.1	55.0
45	630	13898	23377	34672	1.0	22.1	37.1	55.0
50	690	15390	25942	40747	1.0	22.3	37.6	59.1
55	742	16852	28552	41663	1.0	22.7	38.5	56.1
60	818	18127	31087	45795	1.0	22.2	38.0	56.0
65	870	19718	33645	50384	1.0	22.7	38.7	57.9
70	930	21240	36225	55312	1.0	22.8	39.0	59.5
75	990	22688	38813	60232	1.0	22.9	39.2	60.8
80	1050	24179	41377	64403	1.0	23.0	39.4	61.3
85	1117	25649	43935	63525	1.0	23.0	39.3	56.9
90	1163	27135	46500	71692	1.0	23.3	40.0	61.6
95	1238	28605	49088	74048	1.0	23.1	39.7	59.8
100	1290	30083	51637	79012	1.0	23.3	40.0	61.2
105	1349	31500	54248	80745	1.0	23.4	40.2	59.9
110	1417	32947	56775	83686	1.0	23.3	40.1	59.1
115	1477	34485	59370	86228	1.0	23.3	40.2	58.4
120	1538	35498	61912	92339	1.0	23.1	40.3	60.0
125	1598	36953	64492	92528	1.0	23.1	40.4	57.9
130	1657	38881	67081	97979	1.0	23.5	40.5	59.1
135	1710	40319	69675	102863	1.0	23.6	40.7	60.2
140	1785	41753	72210	107258	1.0	23.4	40.5	60.1
145	1830	43259	74805	109020	1.0	23.6	40.9	59.6
150	1898	44737	77333	114008	1.0	23.6	40.7	60.1
155	1964	46139	79920	120751	1.0	23.5	40.7	61.5
160	2025	47101	82492	120967	1.0	23.3	40.7	59.7
165	2069	48517	85049	124049	1.0	23.4	41.1	60.0
170	2145	50557	87638	125797	1.0	23.6	40.9	58.6
175	2197	52020	90202	135825	1.0	23.7	41.1	61.8
180	2257	53460	92759	132517	1.0	23.7	41.1	58.7
185	2310	54952	95347	140085	1.0	23.8	41.3	60.6
190	2369	56408	97905	135983	1.0	23.8	41.3	57.4
195	2438	57945	100493	143490	1.0	23.8	41.2	58.9
200	2497	58695	103028	151935	1.0	23.5	41.3	60.8
				moyennes		23.1	39.4	58.7

Environnement V : AMD Athlon 64 3200+, 2GHz, gcc 4.1.2, unité flottante SSE

	Nombre de cycles				Temps d'exécution normalisé			
deg	H	CH_4	QDH	H_{mpfr4}	H	CH_4	QDH	H_{mpfr4}
10	137	2138	3201	3587	1.0	15.6	23.4	26.2
15	180	3113	4796	5321	1.0	17.3	26.6	29.6
20	219	4066	6320	7042	1.0	18.6	28.9	32.2
25	261	4987	7877	8297	1.0	19.1	30.2	31.8
30	299	5941	9440	9845	1.0	19.9	31.6	32.9
35	337	6862	10992	11639	1.0	20.4	32.6	34.5
40	379	7816	12555	13233	1.0	20.6	33.1	34.9
45	419	8737	14107	15037	1.0	20.9	33.7	35.9
50	458	9691	15665	16707	1.0	21.2	34.2	36.5
55	498	10613	17217	18441	1.0	21.3	34.6	37.0
60	541	11566	18785	20279	1.0	21.4	34.7	37.5
65	579	12487	20344	25710	1.0	21.6	35.1	44.4
70	621	13441	21895	28756	1.0	21.6	35.3	46.3
75	668	14362	23457	30434	1.0	21.5	35.1	45.6
80	698	15316	25006	31273	1.0	21.9	35.8	44.8
85	752	16237	26567	33265	1.0	21.6	35.3	44.2
90	794	17191	28130	35279	1.0	21.7	35.4	44.4
95	834	18113	29682	37671	1.0	21.7	35.6	45.2
100	871	19066	31240	40092	1.0	21.9	35.9	46.0
105	912	19987	32802	40891	1.0	21.9	36.0	44.8
110	951	20941	34355	43801	1.0	22.0	36.1	46.1
115	991	21862	35917	45311	1.0	22.1	36.2	45.7
120	1034	22816	37470	40087	1.0	22.1	36.2	38.8
125	1074	23737	39027	48955	1.0	22.1	36.3	45.6
130	1111	24691	40581	52728	1.0	22.2	36.5	47.5
135	1151	25613	42149	54365	1.0	22.3	36.6	47.2
140	1192	26566	43700	56610	1.0	22.3	36.7	47.5
145	1234	27487	45262	57943	1.0	22.3	36.7	47.0
150	1272	28441	46853	60380	1.0	22.4	36.8	47.5
155	1312	29362	48372	62942	1.0	22.4	36.9	48.0
160	1351	30316	49933	63794	1.0	22.4	37.0	47.2
165	1391	31237	51494	63561	1.0	22.5	37.0	45.7
170	1431	32191	53050	65113	1.0	22.5	37.1	45.5
175	1471	33113	54609	69423	1.0	22.5	37.1	47.2
180	1514	34066	56165	60120	1.0	22.5	37.1	39.7
185	1551	34987	57717	59562	1.0	22.6	37.2	38.4
190	1594	35941	59310	73857	1.0	22.5	37.2	46.3
195	1634	36862	60843	75875	1.0	22.6	37.2	46.4
200	1671	37816	62390	78233	1.0	22.6	37.3	46.8
				moyennes		21.4	34.9	42.0

Environnement VI : Intel Itanium 2, 1.5 GHz, `gcc 4.1.1`

	Nombre de cycles				Temps d'exécution normalisé			
deg	H	CH_4	QDH	H_{mpfr4}	H	CH_4	QDH	H_{mpfr4}
10	191	1635	2969	4834	1.0	8.6	15.5	25.3
15	224	2290	4398	6612	1.0	10.2	19.6	29.5
20	256	2945	5813	8989	1.0	11.5	22.7	35.1
25	269	3600	7214	11049	1.0	13.4	26.8	41.1
30	303	4255	8629	12953	1.0	14.0	28.5	42.7
35	327	4910	10044	15218	1.0	15.0	30.7	46.5
40	359	5569	11472	17439	1.0	15.5	32.0	48.6
45	383	6224	12874	19606	1.0	16.3	33.6	51.2
50	406	6879	14303	20970	1.0	16.9	35.2	51.7
55	439	7534	15717	23215	1.0	17.2	35.8	52.9
60	463	8189	17119	25909	1.0	17.7	37.0	56.0
65	484	8844	18534	27442	1.0	18.3	38.3	56.7
70	518	9499	19949	29509	1.0	18.3	38.5	57.0
75	542	10158	21377	32508	1.0	18.7	39.4	60.0
80	574	10813	22792	33353	1.0	18.8	39.7	58.1
85	598	11468	24208	34594	1.0	19.2	40.5	57.8
90	621	12123	25623	36923	1.0	19.5	41.3	59.5
95	654	12778	27038	39831	1.0	19.5	41.3	60.9
100	678	13433	28439	41611	1.0	19.8	41.9	61.4
105	699	14092	29868	42895	1.0	20.2	42.7	61.4
110	733	14747	31283	46920	1.0	20.1	42.7	64.0
115	757	15414	32698	50176	1.0	20.4	43.2	66.3
120	789	16069	34099	49389	1.0	20.4	43.2	62.6
125	813	16724	35514	53260	1.0	20.6	43.7	65.5
130	834	17379	36929	52800	1.0	20.8	44.3	63.3
135	869	18036	38344	56668	1.0	20.8	44.1	65.2
140	890	18693	39759	59125	1.0	21.0	44.7	66.4
145	914	19348	41174	60310	1.0	21.2	45.0	66.0
150	948	20003	42603	61918	1.0	21.1	44.9	65.3
155	972	20658	44004	65253	1.0	21.3	45.3	67.1
160	1004	21313	45433	65389	1.0	21.2	45.3	65.1
165	1028	21968	46834	69401	1.0	21.4	45.6	67.5
170	1051	22627	48249	70868	1.0	21.5	45.9	67.4
175	1084	23282	49664	70720	1.0	21.5	45.8	65.2
180	1108	23937	51079	77058	1.0	21.6	46.1	69.5
185	1129	24592	52494	82615	1.0	21.8	46.5	73.2
190	1163	25247	53909	80032	1.0	21.7	46.4	68.8
195	1187	25898	55338	81528	1.0	21.8	46.6	68.7
200	1219	26561	56739	84786	1.0	21.8	46.5	69.6
				moyennes		18.7	39.4	58.5

Environnement VII : Intel Itanium 2, 1.5 GHz, `icc 9.1`

	Nombre de cycles				Temps d'exécution normalisé			
deg	H	CH_4	QDH	H_{mpfr4}	H	CH_4	QDH	H_{mpfr4}
10	386	884	3605	4936	1.0	2.3	9.3	12.8
15	406	1044	5240	6740	1.0	2.6	12.9	16.6
20	434	1204	6875	9388	1.0	2.8	15.8	21.6
25	457	1364	8510	10954	1.0	3.0	18.6	24.0
30	474	1524	10145	12912	1.0	3.2	21.4	27.2
35	506	1684	11780	14977	1.0	3.3	23.3	29.6
40	514	1844	13415	17369	1.0	3.6	26.1	33.8
45	537	2004	15050	19946	1.0	3.7	28.0	37.1
50	554	2164	16685	21541	1.0	3.9	30.1	38.9
55	577	2324	18320	23133	1.0	4.0	31.8	40.1
60	594	2485	19957	25382	1.0	4.2	33.6	42.7
65	617	2644	21590	27731	1.0	4.3	35.0	44.9
70	634	2804	23225	28320	1.0	4.4	36.6	44.7
75	666	2964	24860	31128	1.0	4.5	37.3	46.7
80	674	3124	26495	33155	1.0	4.6	39.3	49.2
85	697	3284	28130	34500	1.0	4.7	40.4	49.5
90	714	3444	29765	37802	1.0	4.8	41.7	52.9
95	737	3604	31400	38142	1.0	4.9	42.6	51.8
100	754	3764	33035	41506	1.0	5.0	43.8	55.0
105	777	3924	34670	43445	1.0	5.1	44.6	55.9
110	794	4084	36305	46590	1.0	5.1	45.7	58.7
115	826	4244	37940	49602	1.0	5.1	45.9	60.1
120	834	4404	39575	52314	1.0	5.3	47.5	62.7
125	857	4565	41212	51094	1.0	5.3	48.1	59.6
130	874	4724	42845	53373	1.0	5.4	49.0	61.1
135	897	4884	44480	56297	1.0	5.4	49.6	62.8
140	914	5044	46115	57918	1.0	5.5	50.5	63.4
145	937	5204	47750	59416	1.0	5.6	51.0	63.4
150	954	5364	49385	67369	1.0	5.6	51.8	70.6
155	986	5524	51020	61994	1.0	5.6	51.7	62.9
160	994	5686	52655	65257	1.0	5.7	53.0	65.7
165	1017	5844	54290	68657	1.0	5.7	53.4	67.5
170	1034	6004	55925	69956	1.0	5.8	54.1	67.7
175	1057	6164	57560	72884	1.0	5.8	54.5	69.0
180	1074	6324	59195	77163	1.0	5.9	55.1	71.8
185	1097	6484	60830	77533	1.0	5.9	55.5	70.7
190	1114	6644	62465	77517	1.0	6.0	56.1	69.6
195	1146	6804	64100	77993	1.0	5.9	55.9	68.1
200	1154	6964	65735	82738	1.0	6.0	57.0	71.7
				moyennes		4.8	41.0	51.8

Performances de CompTRSV

Environnement II : Intel Pentium 4, 3.00GHz, `gcc 4.1.2`, unité flottante SSE

n	TRSV	CompTRSV	BLAS_dtrsv_x	n	TRSV	CompTRSV	BLAS_dtrsv_x
10	908	4711	9450	760	2237498	17328540	38387048
20	2191	14903	30690	770	2291423	17761470	39422093
30	3953	31305	64980	780	2354618	18231780	40467480
40	6308	53588	112275	790	2400473	18691455	41503306
50	9308	81968	172905	800	2455493	19200541	42559208
60	13440	117526	246511	810	2513738	19658926	43634011
70	17513	155835	332798	820	2569515	20150378	44721240
80	22193	201773	432345	830	2624371	20606550	45787185
90	27840	253335	545235	840	2686748	21161535	46895550
100	33780	313613	669945	850	2745360	21631058	48040290
110	39135	372128	808808	860	2818193	22157573	49224638
120	45901	441623	959468	870	2870581	22678073	50389305
130	54330	514988	1124303	880	2938598	23286638	51538163
140	62588	596858	1301550	890	2988570	23755478	52753920
150	71798	679358	1492433	900	3046388	24270488	53900160
160	79560	782078	1694986	910	3094253	24807571	55096650
170	90030	870188	1913761	920	3193906	25424731	56324558
180	99376	980528	2143178	930	3268823	25927710	57579105
190	110731	1081561	2385975	940	3314280	26477985	58864448
200	121598	1198081	2641193	950	3372946	27071295	60143483
210	134400	1317166	2910000	960	3443648	27723705	61405433
220	146505	1443811	3191198	970	3505013	28208581	62679555
230	159571	1572308	3486045	980	3562381	28792005	63994988
240	170985	1713375	3792915	990	3646500	29424811	65308935
250	185820	1855823	4114110	1000	3728288	30089580	66662836
260	200228	2002185	4447298	1010	3774473	30678848	68034465
270	215423	2157533	4795283	1020	3846023	31250138	69416070
280	228578	2329666	5160323	1030	3925396	31894403	70736385
290	253095	2489753	5534153	1040	4002173	32566943	72089033
300	275933	2672790	5920771	1050	4079183	33097651	73489388
310	291368	2836748	6318586	1060	4124011	33728685	74960265
320	317168	3028875	6731483	1070	4190205	34399005	76389046
330	339466	3223673	7172738	1080	4308788	35141730	77803793
340	365356	3424718	7611548	1090	4369111	35702010	79178925
350	397358	3628598	8075303	1100	4428061	36256493	80655240
360	407401	3840698	8539321	1110	4501733	36989415	82174898
370	460943	4068368	9033053	1120	4625018	37770638	83654385
380	489826	4305136	9536866	1130	4712678	38351565	85168906
390	513121	4519823	10047271	1140	4743226	38964376	86655376
400	560438	4765448	10565101	1150	4808806	39700238	88232738
410	585293	5006130	11106593	1160	4938968	40537478	89775915
420	631216	5268241	11673698	1170	4998413	41111843	91338248
430	690263	5525895	12261376	1180	5098981	41838188	92834933
440	688463	5770156	12807526	1190	5154796	42492375	94488510
450	752153	6050633	13425945	1200	5297280	43369440	96080933
460	796230	6334321	14022196	1210	5343548	43970985	97721078
470	856051	6606443	14652008	1220	5418285	44711948	99310553
480	885751	6902153	15274013	1230	5484271	45435338	100954216
490	933556	7175003	15920581	1240	5617238	46349543	102608295
500	984345	7500315	16587968	1250	5670646	46977045	104298225
510	1034513	7785510	17274458	1260	5739721	47605156	105916448
520	1068713	8127038	17953298	1270	5854853	48501953	107670211
530	1123471	8437178	18653408	1280	5804146	49310640	109445093
540	1179915	8765258	19380308	1290	6051638	50035081	111013403
550	1206323	9065356	20099926	1300	6119431	50693806	112805183
560	1246718	9421238	20831010	1310	6187125	51588593	114536288
570	1297913	9749355	21583470	1320	6351743	52522636	116216580
580	1355881	10096725	22344173	1330	6393270	53088848	117998528
590	1392968	10437046	23124196	1340	6485528	53849528	119850458
600	1448266	10809405	23908800	1350	6598898	54742013	121633268
610	1485990	11160751	24697973	1360	6758753	55752773	123362588
620	1540943	11525888	25520040	1370	6790501	56408078	125274323
630	1589498	11928023	26345640	1380	6872873	57130200	127119030
640	1604296	12248640	27148463	1390	6967755	57999390	128832893
650	1677068	12657293	28059128	1400	7132261	59090543	130821465
660	1730671	13059083	28921155	1410	7179533	59746260	132780638
670	1761488	13445888	29821051	1420	7306531	60459053	134585783
680	1824751	13888995	30705930	1430	7374158	61449196	136530308
690	1874078	14266838	31632098	1440	7576291	62527643	138485033
700	1928296	14687761	32545050	1450	7580926	63111638	140298608
710	1990185	15100958	33492833	1460	7663875	63991463	142295775
720	2029290	15539828	34439131	1470	7783261	64930080	144137776
730	2071013	15995723	35417168	1480	7964108	65999251	146270640
740	2121571	16391873	36403395	1490	7983781	66697171	148210928
750	2179568	16881195	37381058	1500	8099798	67511843	150215460

Environnement IV : Intel Pentium 4, 3.00GHz, `icc 9.1`, unité flottante SSE

n	TRSV	CompTRSV	BLAS_dtrsv_x	n	TRSV	CompTRSV	BLAS_dtrsv_x
10	825	4028	8258	760	2405093	15055861	33317318
20	2498	12953	26693	770	2464665	15457073	34215638
30	4425	26903	56798	780	2528453	15874343	35098380
40	7035	45578	97996	790	2584043	16266068	36000908
50	10080	69653	150758	800	2645198	16663081	36900810
60	13598	98648	214973	810	2706330	17097735	37830218
70	18165	132623	290220	820	2768161	17519513	38768236
80	22365	171301	376688	830	2828723	17952165	39738061
90	28268	215176	474413	840	2895188	18367456	40659601
100	34755	263805	583673	850	2958323	18819255	41643458
110	41903	317543	704153	860	3031981	19275053	42610268
120	47768	376088	835733	870	3090023	19715198	43601558
130	57315	440761	979845	880	3167550	20151915	44606738
140	65595	509453	1135021	890	3224325	20630730	45648061
150	75113	582345	1298910	900	3303758	21113663	46652558
160	83130	660766	1475663	910	3360923	21591263	47722260
170	95685	745710	1668893	920	3425888	22031266	48753773
180	106988	835283	1864823	930	3506206	22561425	49834238
190	120053	929955	2081918	940	3577583	23040308	50895721
200	129503	1028566	2300296	950	3651938	23529443	51981121
210	144615	1131458	2531978	960	3725123	24077723	53028315
220	157666	1241198	2784353	970	3799080	24571950	54179311
230	172965	1354763	3036961	980	3869123	25073056	55259311
240	185491	1472498	3300541	990	3939526	25588823	56426258
250	202290	1596908	3584956	1000	4013303	26074313	57587708
260	217793	1725646	3870608	1010	4095916	26656170	58700371
270	244058	1866375	4180988	1020	4164451	27186383	59878283
280	249585	1998098	4489553	1030	4249988	27731685	61016573
290	284243	2150565	4824165	1040	4340948	28169701	62220053
300	289958	2292465	5152395	1050	4411741	28811056	63443926
310	325418	2454263	5502308	1060	4481138	29364608	64678606
320	341288	2613833	5862278	1070	4413383	29897753	65817810
330	365101	2778278	6236640	1080	4647060	30359093	67058670
340	406493	2962283	6633825	1090	4741095	31020856	68309896
350	414391	3127666	7026706	1100	4657875	31572638	69569093
360	439793	3319951	7444860	1110	4902008	32169706	70880078
370	475613	3498676	7848061	1120	5020230	32681783	72177008
380	536363	3717593	8298098	1130	5085525	33338093	73390381
390	554108	3909496	8742098	1140	5160683	33903758	74667495
400	584760	4118205	9205861	1150	5263635	34515780	76035818
410	630458	4325986	9667426	1160	5352975	35031848	77352436
420	663068	4539301	10160130	1170	5448278	35762970	78724103
430	718478	4773556	10664925	1180	5519746	36347400	80001916
440	744031	5002951	11152335	1190	5636543	36971730	81384758
450	804075	5246056	11689471	1200	5741146	37469881	82720838
460	860866	5500770	12232680	1210	5820218	38217451	84138458
470	889133	5736893	12768113	1220	5894956	38871045	85545900
480	936090	5986253	13305121	1230	6016081	39518603	86944036
490	1002938	6242693	13890218	1240	5979331	40005931	88352881
500	1051853	6507098	14461020	1250	6209221	40815481	89767388
510	1103565	6782603	15057968	1260	6265725	41469015	91259296
520	1156613	7054950	15652643	1270	6405856	42158708	92610075
530	1190565	7313423	16250333	1280	6464745	43495808	93949096
540	1259393	7618186	16875826	1290	6606083	43468080	95563155
550	1306095	7889453	17495543	1300	6653258	44135753	97094311
560	1357020	8182223	18160793	1310	6821963	44848178	98520218
570	1399485	8470538	18833325	1320	6914963	45289193	100041976
580	1461166	8784736	19474621	1330	7028318	46185061	101557981
590	1497293	9079726	20131433	1340	7072358	46864740	103144725
600	1550671	9390308	20811128	1350	7244760	47583015	104632808
610	1597651	9694253	21517695	1360	7360666	48082868	106188406
620	1651545	10021373	22208588	1370	7466925	49018365	107724653
630	1700903	10345365	22950031	1380	7489328	49717320	109317570
640	1663298	10634003	23631165	1390	7698226	50467748	110919855
650	1799865	10992323	24413438	1400	7765013	50955901	112478955
660	1850941	11364420	25169468	1410	7916768	51933338	114175846
670	1902818	11699281	25932390	1420	7913708	52610790	115708733
680	1957253	12058763	26701756	1430	8143725	53387565	117366383
690	2012303	12404070	27517801	1440	8278186	53913323	119086831
700	2067331	12772980	28295453	1450	8363918	54898035	120679951
710	2122598	13137615	29109038	1460	8358735	55591950	122257546
720	2182666	13526273	29923403	1470	8576783	56394023	124044735
730	2231543	13895926	30773678	1480	8670645	56915693	125712571
740	2296268	14289991	31596608	1490	8817271	57953318	127384028
750	2348356	14671283	32456213	1500	8805458	58694483	129135368

Environnement V : AMD Athlon 64 3200+, 2GHz, gcc 4.1.2, unité flottante SSE

n	TRSV	CompTRSV	BLAS_dtrsv_x	n	TRSV	CompTRSV	BLAS_dtrsv_x
10	578	2563	5454	760	2568565	10469145	24722797
20	1663	8772	18951	770	2653923	10825512	25413049
30	3148	18387	40557	780	2729329	11016945	26044981
40	5050	31310	70584	790	2827209	11401466	26732645
50	7328	47701	108912	800	2845032	11591583	27390322
60	10061	67482	155612	810	2923885	11991027	28107598
70	13122	90507	210582	820	3001736	12168748	28766891
80	16583	117054	273951	830	3076773	12567992	29492785
90	20529	147114	345691	840	3168570	12771023	30178457
100	25125	180692	425729	850	3238825	13173187	30934340
110	29850	218171	514025	860	3319127	13372693	31624752
120	34932	258333	610677	870	3453434	13796653	32408491
130	41212	302952	715881	880	3421689	13994836	33108384
140	47008	350019	829576	890	3522260	14431482	33895588
150	53748	401407	951027	900	3615925	14634375	34627033
160	59974	456023	1081678	910	3671329	15086127	35416110
170	67215	513212	1219813	920	3731418	15277934	36172960
180	75438	574448	1366513	930	3935803	15744599	36992480
190	83491	638660	1520569	940	3910119	15946949	37764097
200	91139	705915	1685255	950	4008405	16426941	38607233
210	100628	778165	1857246	960	4061336	16634197	39363179
220	109968	852388	2037129	970	4198011	17119907	40234758
230	119549	930632	2223559	980	4303674	17307543	41026926
240	128433	1011060	2421598	990	4367462	17819421	41901057
250	139988	1097180	2627499	1000	4421847	18035376	42711348
260	150858	1184832	2840749	1010	4517755	18541433	43621813
270	163070	1277517	3060313	1020	4655566	18744884	44451145
280	172718	1371135	3292931	1030	4775449	19281488	45345390
290	193055	1472728	3533969	1040	4760780	19497779	46182217
300	207385	1575015	3780734	1050	4900389	20026259	47113453
310	230321	1683622	4037124	1060	4956634	20225391	47988435
320	271601	1807603	4331578	1070	5043345	20783278	48920965
330	339392	1930202	4604477	1080	5136074	21013362	49789981
340	353373	2040660	4887956	1090	5273396	21554247	50757738
350	435056	2193980	5198294	1100	5346336	21763270	51629593
360	469706	2322327	5519958	1110	5428429	22352494	52626385
370	531315	2467456	5831128	1120	5512417	22563998	53529901
380	611284	2611131	6174109	1130	5683057	23157004	54547201
390	674197	2780821	6520285	1140	5733254	23373762	55464954
400	722080	2903719	6856458	1150	5818474	23967905	56476664
410	751621	3058206	7191306	1160	5906271	24199361	57402481
420	820690	3205929	7572940	1170	6054698	24809263	58458752
430	888949	3404550	7953687	1180	6133783	25015221	59399397
440	914857	3546541	8312370	1190	6228670	25654435	60443461
450	968198	3733839	8709560	1200	6338045	25868979	61399244
460	940533	3867222	9093719	1210	6449701	26511105	62499949
470	1049232	4054413	9494870	1220	6568018	26721591	63462409
480	1086519	4197252	9895337	1230	6624502	27386913	64556821
490	1144325	4413854	10320067	1240	6727195	27589009	65553133
500	1114744	4567457	10738977	1250	6876000	28240365	66676885
510	1233429	4784463	11179139	1260	6973359	28492926	67691869
520	1268490	4939658	11604184	1270	7085861	29181343	68816365
530	1331192	5176722	12071813	1280	7077776	29377643	69740623
540	1372963	5318821	12511557	1290	7305597	30105003	70998208
550	1387366	5556528	12994752	1300	7404703	30307051	72026869
560	1451598	5709627	13461195	1310	7497838	31029970	73194829
570	1538132	5973487	13954370	1320	7598485	31247489	74230105
580	1529042	6129742	14427053	1330	7762440	31978150	75438516
590	1642615	6398850	14944318	1340	7866729	32189380	76496893
600	1606904	6553924	15443804	1350	7953744	32934231	77698747
610	1710686	6826231	15969807	1360	8068509	33138326	78762301
620	1744385	7003795	16477290	1370	8218903	33917891	80024173
630	1860493	7269924	17032737	1380	8318518	34120217	81113862
640	1887879	7436594	17549329	1390	8439326	34889448	82358173
650	1979690	7735335	18142848	1400	8533539	35094115	83456233
660	1985757	7915318	18676177	1410	8710910	35913354	84732316
670	2088672	8226849	19253084	1420	8802233	36101320	85866078
680	2070451	8406216	19798737	1430	8901325	36919695	87137846
690	2220101	8726786	20425851	1440	9005291	37144336	88271953
700	2212593	8886395	20996762	1450	9179947	37959540	89582922
710	2315159	9232521	21606811	1460	9300358	38153658	90744997
720	2335794	9413337	22202168	1470	9404851	38999291	92062993
730	2377927	9756073	22856257	1480	9532985	39206168	93229495
740	2483230	9936602	23464369	1490	9702623	40073284	94588681
750	2523694	10292789	24113850	1500	9795175	40263402	95756588

Environnement VI : Intel Itanium 2, 1.5 GHz, `gcc 4.1.1`

n	TRSV	CompTRSV	BLAS_dtrsv_x	n	TRSV	CompTRSV	BLAS_dtrsv_x
10	257	721	1770	760	524239	2027219	7521881
20	663	1981	5914	770	545055	2056949	7721227
30	1224	3991	13196	780	552420	2110412	7922630
40	1898	6402	21918	790	571685	2164792	8126851
50	2792	9668	33767	800	579974	2402203	8333359
60	3851	13544	48212	810	602125	2274581	8542937
70	5001	18136	65232	820	609018	2330866	8755273
80	6260	23048	84820	830	630023	2446604	9389284
90	7858	29215	106988	840	638793	2473997	9186505
100	9595	35709	131733	850	662380	2504327	9406066
110	11342	42971	159052	860	669819	2563152	9628803
120	13123	50119	188939	870	691540	2622501	9853263
130	15433	59223	221430	880	699786	2776968	10080266
140	17543	68497	256498	890	725480	2743850	10311470
150	20219	78340	294094	900	732658	2806074	10543907
160	22842	97198	334304	910	755746	2867831	10779610
170	25574	99876	377141	920	764580	2965096	11016692
180	28170	111742	423275	930	792106	2994948	11257663
190	31572	127605	493794	940	799101	3059292	11500403
200	34526	139229	521936	950	823591	3124781	11746321
210	38338	151252	574953	960	832174	3453994	11994057
220	41939	166479	631441	970	861296	3257591	12246090
230	45738	180980	689175	980	869281	3325720	12500478
240	51345	207191	752053	990	893382	3475375	13352798
250	55657	215820	816423	1000	903363	3501905	13014326
260	60378	234344	883368	1010	932320	3530471	13275352
270	65955	253101	952331	1020	940228	3601374	13542084
280	70558	275824	1024326	1030	975331	3681257	13815073
290	77305	293194	1098940	1040	978632	3877352	14078547
300	82108	314178	1175881	1050	1010828	3818419	14350265
310	89020	336076	1255866	1060	1035906	3911923	14645501
320	93539	387889	1337837	1070	1054833	3974394	14911537
330	101054	380492	1422920	1080	1061698	4093216	15187907
340	106336	404704	1509894	1090	1087518	4113049	15463826
350	113920	439414	1675388	1100	1132193	4227709	15787566
360	119194	458747	1693056	1110	1131230	4274089	16046274
370	127630	479424	1788223	1120	1153800	4719290	16344439
380	133255	505447	1885768	1130	1269355	4539073	16748644
390	141459	532132	1986412	1140	1199658	4525027	16948504
400	147044	578634	2088824	1150	1251664	4737599	18073220
410	156663	587521	2194337	1160	1259047	4753154	17553228
420	162514	616451	2302369	1170	1579260	5078173	18164563
430	171701	645782	2413326	1180	1303747	4862693	18174558
440	177872	683561	2526357	1190	1370163	4984707	18512960
450	188297	706814	2642070	1200	1393522	5253573	18846583
460	194373	738361	2760489	1210	1395310	5127808	19117521
470	204653	770710	2881748	1220	1394063	5210103	19435826
480	211009	869167	3004853	1230	1553402	5422874	19887855
490	222665	837594	3131836	1240	1505734	5499769	20133964
500	229210	872193	3261117	1250	1614295	5609437	20553273
510	240462	929129	3550964	1260	1881434	5964343	21149464
520	247633	953345	3526174	1270	1603736	5714929	21135669
530	260464	979209	3663138	1280	1688613	6362165	21555589
540	267191	1016140	3802278	1290	1847405	6094957	22009963
550	279487	1053727	3943924	1300	1788227	6114037	22277652
560	286374	1129488	4088226	1310	2074689	6606597	23913316
570	300693	1131035	4235435	1320	1832449	6365201	22954102
580	307477	1171001	4385538	1330	1939860	6446328	23361122
590	320922	1211606	4537361	1340	1957485	6551875	23712945
600	328432	1266985	4691962	1350	1903736	6562969	23988104
610	343834	1294460	4849750	1360	2049726	7014041	24473711
620	350825	1337039	5009616	1370	2122833	6901715	24841207
630	365095	1380210	5172486	1380	2000608	6879983	25096993
640	372901	1541019	5336977	1390	2056784	6989443	25467362
650	390081	1468199	5504951	1400	2341768	7431561	26088011
660	397340	1513722	5675713	1410	2391683	7467817	26488186
670	412910	1598100	6122315	1420	2397267	7576303	26872116
680	420979	1624955	6024268	1430	2468542	7685793	27241108
690	438858	1654091	6202268	1440	2677847	8560428	27801893
700	446285	1701920	6383428	1450	2592971	7959035	28106689
710	463099	1750319	6566485	1460	2735545	8184111	28611655
720	470891	1862232	6752137	1470	2753741	8474116	30325732
730	490539	1850023	6941008	1480	3004369	8685705	29588420
740	497832	1900499	7132133	1490	3404484	9013125	30273061
750	515953	1951875	7325845	1500	3210298	8958307	30504174

Résumé : Les erreurs d'arrondi peuvent dégrader la précision d'un calcul en arithmétique flottante. Comment améliorer et valider la précision d'un résultat calculé, tout en conservant de bonnes performances pratiques ? Nous utilisons la compensation des erreurs d'arrondi au travers de deux exemples : l'évaluation polynomiale et la résolution de systèmes linéaires triangulaires. Nos contributions se situent à trois niveaux.

1) *Amélioration de la précision du résultat.* Nous proposons un schéma de Horner compensé, qui permet une évaluation polynomiale aussi précise que celle calculée par le schéma de Horner classique exécuté avec une précision interne doublée. En généralisant cet algorithme, nous proposons une autre version compensée du schéma de Horner simulant K fois la précision de travail ($K \geq 2$). Nous montrons également comment compenser les erreurs d'arrondis générées par l'algorithme de substitution pour la résolution de systèmes triangulaires.

2) *Validation de la qualité du résultat.* Nous montrons comment valider la qualité du résultat de l'évaluation polynomiale compensée, en proposant le calcul d'une borne d'erreur *a posteriori* qui ne repose que sur des opérations élémentaires de l'arithmétique flottante : cela assure la portabilité de la validation et de bonnes performances pratiques.

3) *Performances des algorithmes compensés.* Nos mesures de performances montrent l'intérêt pratique des algorithmes compensés face aux autres solutions logicielles simulant une précision équivalente. Nous justifions ces bonnes performances par une étude détaillée du parallélisme d'instructions qu'ils présentent.

Mots-clés : Arithmétique des ordinateurs, norme IEEE-754, algorithmes numériques, algorithmes compensés, performances.

Abstract: Rounding error may totally corrupt the result of a floating point computation. How to improve and validate the accuracy of a floating point computation, without large computing time overheads ? We contribute to this question considering two examples: polynomial evaluation and linear triangular system solving. In both cases we use the compensation of the rounding errors to improve the accuracy of the computed results.

1) *Improving the accuracy.* We propose a compensated Horner scheme that computes polynomial evaluation with the same accuracy as the classic Horner algorithm performed in twice the working precision. Generalizing this algorithm, we present another compensated version of the Horner scheme simulating K times the working precision ($K \geq 2$). We also show how to compensate the rounding errors generated by the substitution algorithm for triangular system solving.

2) *Validating the computed result.* We show how to validate the quality of the compensated polynomial evaluation. We propose a method to compute an *a posteriori* error bound together with the compensated result. This error bound is computed using only basic floating point operations to ensure portability and efficiency of the method.

3) *Performances of compensated algorithms.* Our computing time measures show the interest of compensated algorithms compared to other software solutions that provide the same output accuracy. We also justify such good practical performances thanks to a detailed study of the instruction-level parallelism they contain.

Keywords: Computer arithmetic, IEEE-754 standard, numerical algorithms, compensated algorithms, efficient algorithms.